HOCHSCHULE
MITTWEIDA
UNIVERSITY OF
APPLIED SCIENCES

BACHELORARBEIT

Herr
Claudio Heeg

Nutzen von domainspezifischen Sprachen in bestehender Softwareentwicklung zur Analyse- und Wartungsverbesserung

Mittweida, 2013

Fakultät Mathematik/
Naturwissenschaften/Informatik

BACHELORARBEIT

Nutzen von domainspezifischen Sprachen in bestehender Softwareentwicklung zur Analyse- und Wartungsverbesserung

Autor:
Herr Claudio Heeg

Studiengang:
Informatik

Seminargruppe:
IF10W1-B

Erstprüfer:
Prof. Dr. Rudolf Stübner

Zweitprüfer:
Dipl.-Inf. Sven Klemm

Einreichung:
Mittweida, 28.08.2013

Verteidigung/Bewertung:
Mittweida, 2013

Bibliografische Beschreibung:

Heeg, Claudio:
Nutzen von domainspezifischen Sprachen in bestehender Softwareentwicklung zur Analyse- und Wartungsverbesserung. - 2013. - vi, 51, I S.
Mittweida, Hochschule Mittweida, Fakultät Mathematik/Naturwissenschaften/ Informatik, Bachelorarbeit, 2013

Referat:

Die vorliegende Arbeit befaßt sich mit der Entwicklung von Tools zur Wartungs- und Analyseverbesserung mittels domänenspezifischer Sprachen (DSLs). Wichtigstes Ziel ist die Erstellung einer Umgebung, die es ermöglicht, bestehende Software bzw. bestehenden Programmcode zu untersuchen und auf Basis von Hinweisen seitens des Tools eine Qualitätssteigerung zu ermöglichen. Zur Erreichung dessen wird unter Nutzung bestehender Frameworks eine Sprachbeschreibung für die bestehende DSL konzeptuell erstellt und schließlich implementiert. Hierauf aufbauend geschieht die Integration in ein Analysetool, welches es Nutzern ermöglichen soll, lokal geltende Best Practices für die DSL zu spezifizieren und darauf aufbauend quantitative sowie qualitative Codeuntersuchungen durchzuführen.

Danksagung

An dieser Stelle möchte ich mich bei allen Personen bedanken, die mich während meines Studiums und insbesondere während meiner Bachelorarbeit unterstützt haben.

Besonderer Dank gilt an dieser Stelle meinen Eltern Michael und Gerlinde Heeg für die Unterstützung während des gesamten Studiums.

Zusätzlich danke ich meinen Betreuern und Prüfern Dipl.-Inf. Sven Klemm sowie Prof. Dr. Rudolf Stübner für ihre Tipps sowie ihre Bereitschaft, aufkommende Fragen zu den Themen der Arbeit und zur Arbeit selbst zu beantworten.

Nicht zuletzt bedanke ich mich auch bei meinen Freunden, allen voran Michael Jung, Jette Bajorat und Yannic Boida, für die konstruktive Kritik im Verlaufe der Arbeit sowie das Korrekturlesen.

Inhalt

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

API	Application Programming Interface (Programmierschnittstelle)
C#	C-Sharp
CDT	C Development Toolkit
COBOL	Common Business Oriented Language
DCL	Data Control Language (Datenüberwachungssprache)
DDL	Data Definition Language (Datendefinitionssprache)
DML	Data Manipulation Language (Datenmanipulationssprache)
DSL	Domain-Specific Language (domänenspezifische Sprache)
GPL	General Purpose Language (Universelle [Programmier-]Sprache)
IDE	Integrated Development Environment (integrierte Entwicklungsumgebung)
LGPL	Lesser General Public License
LOC	Lines Of Code (Codezeilenanzahl)
NCLOC	Non-Comment Lines Of Code (Nichtkommentar-Codezeilen)
PL/SQL	Procedural Language/SQL (prozedurales SQL)
RDBMS	Relational Database Management System (relationales Datenbankmanagementsystem)
RegEx	Regular Expressions (reguläre Ausdrücke)
SQL	Structured Query Language (Strukturierte Abfragesprache)
TCL	Transaction Control Language (Transaktionskontrollsprache)
UTF	Unicode Transformation Format

1 Übersicht

In diesem einleitenden Kapitel werden Motivation sowie Aufgabenstellung dieser Bachelorarbeit besprochen. Gleichzeitig erfolgt ein kurz gefaßter Überblick über ihre Kapitel.

1.1 Motivation

Aktuelle Programmiertechniken setzen verstärkt domainspezifische Sprachen (DSLs) ein, um den Fokus der Entwicklung auf Fachlichkeit zu lenken und technologische Unabhängigkeit zu gewinnen.

Bestes Beispiel für die Nutzung von DSLs in der Technik ist die Datenbanksprache SQL. Auch reguläre Ausdrücke (*regular expressions*, „RegEx“) sind mittlerweile, als Mittel zur Vereinfachung der Definition von Stringmustern und somit direkt dem Abbau der Codemenge dienlich, Teil vieler Programmiersprachen geworden.

Beispiele für den Nutzen domainspezifischer Sprachen gibt es auch in kleinem Rahmen, unter anderem in der Produktentwicklung eines Softwareunternehmens, viele. So ist es denkbar, daß einem informatiktechnisch weniger versierten Endnutzer in einem Endprodukt eine vereinfachte Version einer Sprache zur Verfügung gestellt wird, um die Benutzung zu erleichtern und Fehlerquellen einzudämmen.

Ein weiterer Aspekt ist auch die firmeninterne Nutzung solcher DSLs, um Softwareentwickler bei der Fehlerfindung sowie Codeanalyse zu unterstützen. Zur Steigerung der Arbeitsleistung und Produktqualität wird es so also zunehmend interessanter, eine auf einen konkreten Problembereich zugeschnittene Sprache zu entwickeln.

1.2 Zielsetzung

Im Rahmen dieser Bachelorarbeit soll für den Einsatz in der Produktentwicklung bei Robotron zunächst mit Hilfe der Eclipse-IDE eine Xtext-Grammatik für die bestehende Sprache PL/SQL entwickelt werden, ergänzt um firmenintern eingeführte Beschränkungen – beispielsweise ist der Einsatz einiger Sprach-Features unerwünscht.

Ergänzend hierzu soll eine Integration in das Application Lifecycle Management, vor allem in das Codeanalysetool Sonar, erfolgen.

Hauptziel ist es, durch die Entwicklung der Xtext-Grammatik eine Kontrollmöglichkeit zu schaffen, um den wachsenden Programmquellcode im Produkt angemessen zu warten.

Zum Zweiten sind auf Basis der Hinweise der Sprachprüfung unter anderem Aussagen bezüglich der Schwere der Fehler sowie verbesserte Analysen möglich – beispielsweise hinsichtlich der Verwendung bestimmter Sprachkonstruktionen.

In einem optionalen dritten Schritt kann die Umwandlung von PL/SQL-Code in andere Sprachen oder deren Interpretation auf anderen Systemen Anwendung finden.

Auf PL/SQL, das Xtext-Framework sowie das Codeanalysetool Sonar wird im Folgekapitel näher eingegangen.

1.3 Kapitelübersicht

Die Bachelorarbeit setzt sich aus fünf Kapiteln zusammen.

Nach der allgemeinen Einleitung mit Darstellung der Motivation sowie Zielstellung in diesem Kapitel werden in **Kapitel 2** die Grundlagen domainspezifischer Sprachen, PL/SQL, des Xtext-Frameworks sowie des Codeanalysetools Sonar erläutert.

Kapitel 3 beinhaltet eine durch ein Pflichtenheft gestützte Anforderungsanalyse, in welcher die Teilziele dieser Bachelorarbeit näher präzisiert und ihr Aufwand sowie die Methoden und Alternativen ihres Erreichens abgeschätzt werden. Zudem werden in diesem Kapitel Lösungskonzeptionen der Teilprobleme entsprechend vorgestellt.

Das sich anschließende **Kapitel 4** befaßt sich mit der Umsetzung der Anforderungen. In diesem Kapitel werden verschiedene Aspekte der Xtext-Grammatik dargestellt, außerdem wird auf die erreichte Verbesserung der Codeanalyse Bezug genommen.

Letztlich werden in **Kapitel 5** die Resultate dieser Bachelorarbeit noch einmal zusammengefaßt und den Zielen gegenübergestellt. Es beinhaltet zudem einen Ausblick auf mögliche zukünftige Weiterentwicklungen und Verbesserungen der erreichten Implementierungen.

2 Grundlagen

Im Folgenden werden die für das Thema dieser Bachelorarbeit relevanten technischen Grundlagen beschrieben. Neben einer Definition des Begriffs der domainspezifischen Sprache werden bestehende Anwendungen und Modelle dargestellt, auf die die Arbeit sich stützt.

Dieses Wissen soll als Basis fungieren, auf welche aufbauend schließlich ein Konzept zur Erreichung der gesetzten Ziele erarbeitet wird.

2.1 DSL – Begriffsdefinition

Für den Begriff der domainspezifischen Sprache existieren verschiedene, sich weitestgehend ähnelnde Definitionen – als Grundlage für die hier aufgestellte Interpretation des Begriffs soll die in [Fowl2010] beschriebene Definition domainspezifischer Sprachen dienen:

> *"**Domain Specific Language (noun):** a computer programming language of limited expressiveness focused on a particular domain."*

Eine DSL ist hiernach also eine „Programmiersprache mit begrenzter Ausdrucksmächtigkeit, die sich auf ein bestimmtes Problemfeld konzentriert".

Es ergibt sich aus dieser kurzen Definition eine Interpretation, die auf drei Schlüsselaspekte fußt:

- DSLs sind – als **Programmiersprachen** – formale Sprachen[1], in deren Vordergrund nicht die Kommunikation, sondern die mathematische Verwendung steht. Sie sollten in diesem Sinne einerseits von Computern ausführbar sein, gleichermaßen jedoch eine menschenverständliche Struktur aufweisen. Ihre Sprachstruktur sollte zudem einen einer Programmiersprache üblichen Fluß aufweisen – das Ausdrucksvermögen der Sprache sollte also nicht nur aus ihren Ausdrücken selbst erwachsen, sondern zusätzlich durch ihre Komposition zueinander.
- Der Aspekt der **begrenzten Ausdrucksmächtigkeit** grenzt DSLs insbesondere von GPLs wie Java und C ab. Während solche universellen Sprachen eine große Menge an Möglichkeiten bezüglich Kontroll- und Datenstrukturen bieten, sollen

[1] weitere grundlegende Informationen zu formalen Sprachen sind in [TLethen06] zu finden

domainspezifische Sprachen lediglich eine minimale Teilmenge dieser Funktionalitäten umfassen, jedoch genug, um der Problemspezifikation des entsprechenden Umfelds gerecht zu werden. Sie beschreiben in diesem Sinne lediglich einen Teil eines sie beinhaltenden größeren Softwaresystems.
- Ein weiterer elementarer Gesichtspunkt in der Definition ist der namensgebende **Domainfokus**. Eine DSL zieht ihren Hauptnutzen aus der Verwendung in einem bestimmten Problembereich – das bedeutet, daß die Konzentration auf eben diesen Bereich für die Sinnhaftigkeit der Sprache besonders wichtig ist. Befehlssatz und Funktionsmenge der DSL sollen sich also am in der Domäne zu lösenden Problem orientieren.

Zusätzlich zu dieser Definition sind domainspezifische Sprachen nach ihrer Art in intern und extern zu unterscheiden:

- **Interne DSLs** (auch: eingebettete DSLs) orientieren sich stark an universellen Programmiersprachen – sie nutzen die Grammatik einer existierenden Sprache unter einem eingeschränkten Befehlsvorrat, um eine Fokussierung auf ein Teilproblem zu erreichen. Das heißt, daß immer eine echte Untermenge der in der GPL zur Verfügung stehenden Sprachkonstrukte verwendet wird. Insbesondere bedeutet das, daß solche DSLs keinen eigens zugeschnittenen Parser oder Interpreter benötigen, also vorhandene Infrastruktur nutzen können. Als Beispiel für diesen Sprachtyp sind auf Lisp basierende DSLs anzuführen.
- **Externe DSLs** sind solche Sprachen, die von Grund auf neu definiert sind. Sie unterscheiden sich oftmals stark von jeglichen existierenden Sprachen und sind sowohl syntaktisch als auch semantisch genau auf die Anwendungsdomäne zugeschnitten – sie besitzen also eine eigene Sprachgrammatik. Die Ausführung des Codes externer DSLs kann einerseits mit einem speziellen Interpreter und andererseits durch die vorherige Umwandlung in eine andere Sprache erfolgen. Beispiele für solche Sprachen sind RegEx sowie SQL.

Diese Interpretation domainspezifischer Sprachen zusammenfassend, heißt das, daß DSLs grundsätzlich als Mittel zur Lösung von Teilproblemen eines Softwaresystems zu betrachten sind. Hierbei ist es in der Praxis nicht unüblich, daß in einem komplexen Softwaresystem in unterschiedlichen Bereichen verschiedene DSLs zur Anwendung kommen.

2.2 PL/SQL

PL/SQL ist die prozedurale Erweiterung der Programmiersprache SQL. Sie wird als proprietärer Zusatz für die Sprache seitens der Oracle Corporation[2] angeboten und unter anderem als Teil von „Oracle Database“ vertrieben.

[2] Webpräsenz des Unternehmens – http://www.oracle.com

Für eine genauere Beschreibung von PL/SQL sind vorher Erläuterungen betreffs ihrer Basissprache SQL zu treffen.

2.2.1 SQL

SQL selbst ist, wie im Vorlauf bereits erwähnt, eine DSL zur Arbeit mit Daten in relationalen Datenbankmanagementsystemen[3] (RDBMS). Als deklarative Programmiersprache beschreibt SQL – in Abgrenzung zu prozeßorientierten Sprachen – befehlsmäßig nicht, wie eine Aufgabe ausgeführt werden soll; ein SQL-Befehl deklariert lediglich die Art der auszuführenden Aufgabe. Die SQL-Sprache ist in vier Subsprachen unterscheidbar:

Data Definition Language (DDL)	Beschreibung/Definition von Datenstrukturen, z.B. `CREATE TABLE, DROP TABLE`
Data Manipulation Language (DML)	Veränderung und Abfrage von Daten innerhalb einer Datenbank, z.B. `SELECT`[4], `INSERT, UPDATE, DELETE`
Data Control Language (DCL)	Steuerung von Benutzerberechtigungen, z.B. `GRANT, REVOKE`
Transaction Control Language (TCL)	Steuerung von Datenbanktransaktionen, z.B. `COMMIT, ROLLBACK`

Tabelle 2-1: *SQL-Subsprachen*

Diese Subsprachen können ihrerseits granular weiter eingeteilt werden – so wird teilweise in der DCL zusätzlich zwischen Session Control sowie System Control unterschieden. Oft unterscheiden sich die Abgrenzungen der Teilsprachen untereinander geringfügig, so erfaßt beispielsweise Oracle selbst alle DCL-Statements als Teil der DDL.

Abbildung 2-2 zeigt einen `SELECT`-Befehl, der aus einer fiktiven Tabelle `Articles` alle Artikel mit einem Preis größer als 42 auswählt und nach Namen sortiert ausgibt.

```
SELECT *
 FROM Articles
 WHERE price > 42.00
 ORDER BY name;
```

Abbildung 2-2: *Beispiel eines simplen SQL-Befehls*

[3] Definition von RDBMS – http://www.itwissen.info/definition/lexikon/relational-database-management-system-RDBMS-Relationales-Datenbank-Managementsystem.html

[4] Das SELECT-Statement wird oftmals auch als Spezialfall in die DQL (Data Query Language) eingeordnet.

Wie aus den Listings ersichtlich ist, sind SQL-Befehle sehr nahe an die natürliche Sprache angelehnt, sodaß ein prinzipielles Verständnis auch für Laien – sofern der englischen Sprache mächtig – problemlos möglich ist. So wäre eine sinngemäße „wörtliche“ Übersetzung des Befehls nach Listing 2-2 an dieser Stelle

```
WÄHLE * (alles) VON Artikel WO preis > 42, SORTIERE NACH name.
```

Wie im vorigen Abschnitt dargestellt, ist ein solches Merkmal typisch für eine domainspezifische Sprache.

Befehle werden in SQL prinzipiell voneinander getrennt behandelt – das bedeutet also, daß jeder Befehl von seinem unmittelbaren Vorgänger und Nachfolger unabhängig ist und eine eigens gekapselte – atomare – Einheit bildet. Weiterhin bieten sich dem Nutzer in SQL keine Möglichkeiten der dynamischen Informationsverarbeitung; insbesondere Dateneingabe zur Laufzeit ist nicht möglich. Aufgrund dieser Gegebenheiten existieren in SQL keine „kompletten“ Programme im üblichen Sinne – jeder Befehl ist als eigenes Programm anzusehen, Befehlsabläufe somit als eine Aneinanderreihung von Programmen.

Zur Arbeit mit Datenbanken in Verbindung mit SQL existiert ein breites Spektrum an Software – so stellt Oracle hierfür die IDE „SQL Developer“[5] kostenfrei zur Verfügung.

2.2.2 PL/SQL als Erweiterung von SQL

PL/SQL setzt als Spracherweiterung auf SQL auf und besitzt erweiterte Funktionalitäten bezüglich der prozeduralen Abarbeitung von Befehlen. Syntax, Struktur sowie Datentypen der Sprache sind sehr stark an die Sprache Ada[6] angelehnt. Es besitzt Merkmale prozeduraler Programmiersprachen, jedoch auch Elemente objektorientierter Programmierung.

Im Gegensatz zum deklarativen SQL beschreibt PL/SQL, wie eine Aufgabe ausgeführt werden soll. Als programmatische Einheiten in PL/SQL können beispielsweise Pakete, Funktionen oder Prozeduren definiert werden.

Die Sprache ist im Gegensatz zu reinem SQL für die Anwendungsentwicklung ausgelegt und bietet so wesentliche Features, die auch in anderen universellen Programmiersprachen üblich sind (entlehnt aus [Fega1998]):

- PL/SQL ermöglicht es, Anweisungen in Blöcken zu beschreiben. Diese Blöcke werden in `BEGIN`- und `END`-Statements gefaßt. Sie können entweder als Unterpro-

[5] Oracle SQL Developer – http://www.oracle.com/technetwork/developer-tools/sql-developer/overview/index.html

[6] Ada, eine strukturierte Programmiersprache – http://www.adaic.org/resources/add_content/standards/95lrm/ARM_HTML/RM-TOC.html

gramme eine Bezeichnung tragen oder als anonyme Blöcke bezeichnungsfrei existieren. Insbesondere SQL-DML-Statements können in solche Blöcke direkt eingebettet werden – eine Einbettung von DDL-Statements macht die Nutzung dynamischer SQL-Statements[7] in PL/SQL-Code notwendig.

- Die Sprache bietet die Möglichkeit, den Programmfluß mit Auswahlen (`IF`, `CASE`), Schleifen (`LOOP`, `WHILE`, `FOR`) und Sprüngen (`GOTO`) zu steuern.
- In PL/SQL besitzt der Programmierer die Möglichkeit, Variablen, Konstanten und Typen zu definieren. Die Notwendigkeit einer solchen Funktionalität ergibt sich auch daraus, daß sich Daten zur Programmlaufzeit ändern oder erst durch eine Nutzereingabe bekannt werden können.
- Weiterhin unterstützt PL/SQL spezielle auf Datenbankmanipulation zugeschnittene Programmstrukturen wie Cursors – zur Laufzeit durch Statements erzeugte, temporäre Arbeitsbereiche – und Trigger – Programmcode, der nur nach der Ausführung bestimmter DML-Statements durchlaufen wird.
- Dem Entwickler wird die Option gegeben, eine eigene Fehlerbehandlung durchzuführen und Fehler entsprechend abzufangen.

Abbildung 2-3 zeigt ein kurzes Beispielprogramm in PL/SQL – es stellt eine kurze Prozedur zur Aufsummierung von Preisen dar.

```
DECLARE                                    -- Variablen- und
  v_summe        NUMBER := 0;              -- Konstantendeklarationen
  v_pos_preis    NUMBER;
  CURSOR c_auftrag_pos IS
    SELECT anzahl*preis FROM auftrag_pos;

BEGIN                                      -- Programmblock
  OPEN c_auftrag_pos;
  LOOP                                     -- Schleife (für jeden
                                           -- Datensatz im Cursor durchlaufen)
    FETCH c_auftrag_pos INTO v_pos_preis;
    EXIT WHEN c_auftrag_pos%NOTFOUND;
    v_summe := v_summe + v_pos_preis;
  END LOOP;
  CLOSE c_auftrag_pos;
END;
```

Abbildung 2-3: *Programmbeispiel in PL/SQL*

Der im vorigen Abschnitt erwähnte „SQL Developer“ kann ebenfalls mit PL/SQL-Code umgehen – tatsächlich ist er sogar eher auf die Arbeit mit PL/SQL abgestimmt, SQL als echte Teilmenge davon betrachtend.

2.3 Eclipse

Eclipse[8] ist ein Programmierwerkzeug zur IDE-gestützten Entwicklung verschiedener Softwareprojekte. Ursprünglich wurde die Anwendung als Entwicklungsumgebung für Ja-

[7] dynamisches SQL – zur Anwendungslaufzeit generierte SQL-Statements

[8] Das Eclipse-Projekt – http://www.eclipse.org/

va-Entwickler konzipiert, mittlerweile bietet sie jedoch zusätzlich Möglichkeiten für die Programmierung anderer Sprachen und Modelle. Die Software besitzt einen modularen Aufbau, das bedeutet, sie ist als Konglomerat von Plugins (auch: Bundles) zu betrachten, die sich auf verschiedene Teilaspekte oder eigene Programmiersprachen beziehen. So existieren für Eclipse unter anderem Bundles zur Erstellung von C-Programmen[9] oder Python-Programmen[10], jedoch auch Tools, die z.B. Abdeckungsstatistiken für Code bieten. Eclipse bietet durch dieses Konzept Entwicklern die Möglichkeit zur individuellen Anpassung.

Als IDE ist das Ziel der Software, die Programmierung zu erleichtern – so umfaßt das Produkt, unter anderem, folgende Funktionalitäten:

- Projektverwaltung aller Projekte im Arbeitsumfeld
- Hervorhebung bestimmter sprachspezifischer Schlüsselwörter
- Markierung von Fehlern und schlechter Praktiken während der Programmierung
- Möglichkeit der programmgestützten Code-Restrukturierung (Refactoring)
- Assistenten (Wizards) zum Export fertiger Softwareprojekte
- Hilfe mit Hinweisen zur Sprachreferenz der jeweiligen Sprache

Listing 2-4 zeigt einen Ausschnitt der aktuellen Version 4.2.2 der Software.

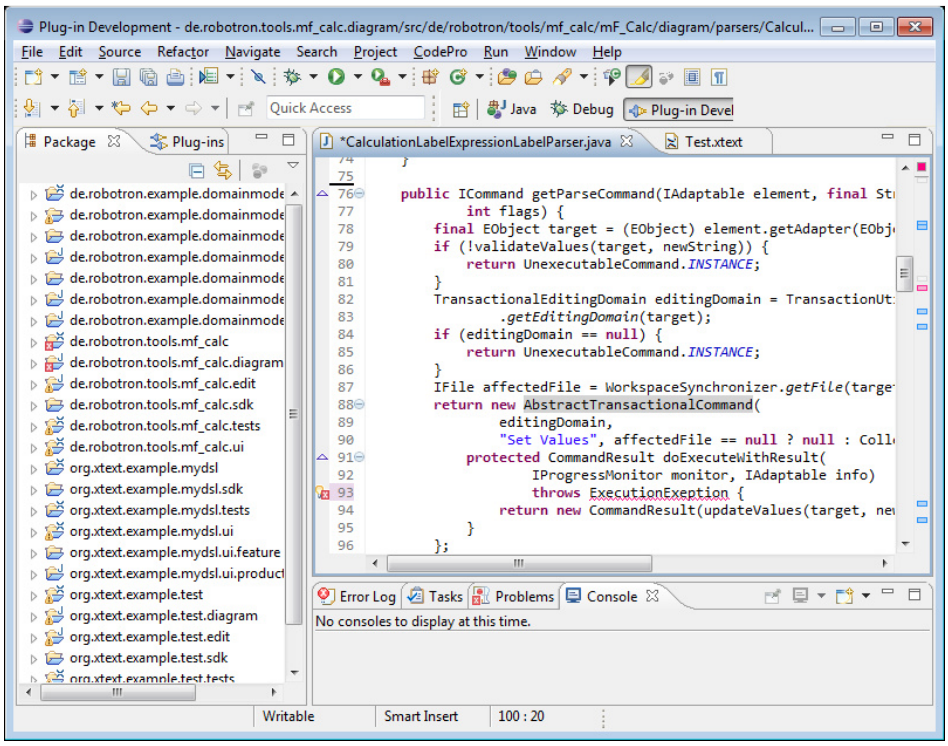

Abbildung 2-4: *Die Eclipse-IDE (Version 4.2.2)*

[9] CDT-Plugin zur C/C++-Entwicklung

[10] PyDev, ein Eclipse-Plugin zur Python-Programmierung – http://pydev.org/

2.4 Xtext

Xtext[11] ist ein kostenloses Open-Source-Tool zur Vereinfachung von Design und Implementierung domainspezifischer Sprachen. Es wurde 2006 im openArchitectureWare[12]-Projekt veröffentlicht und dort bis 2008 entwickelt – seit diesem Zeitpunkt wird Xtext bei Eclipse unter dem „Eclipse Modeling Project“ weiterentwickelt. Derzeit wird das Tool als Eclipse-Plugin hauptsächlich von der deutschen Unternehmung itemis[13] betreut.

Das Xtext-Framework bietet die in die Eclipse-Umgebung eingebettete Möglichkeit zur Erstellung einer speziellen Grammatik für eine bereits existierende oder noch zu erstellende DSL. Zusätzlich bietet das Tool die Möglichkeit, einen auf die im konkreten Fall bearbeitete DSL zugeschnittenen Editor nach dem Vorbild von Eclipse zu erstellen. Dem Entwickler wird die Option gegeben, verschiedene Elemente wie beispielsweise Syntax-Hervorhebung, Auto-Vervollständigung, Auto-Formatierung und Korrekturvorschläge für Fehler auf die DSL zuzuschneiden und im Editor zur Verfügung zu stellen.

Listing 2-5 zeigt beispielhaft einen Editor für eine Domainmodel-DSL mit angepaßter Syntaxhervorhebung und eigenen Fehlermarkierungen.

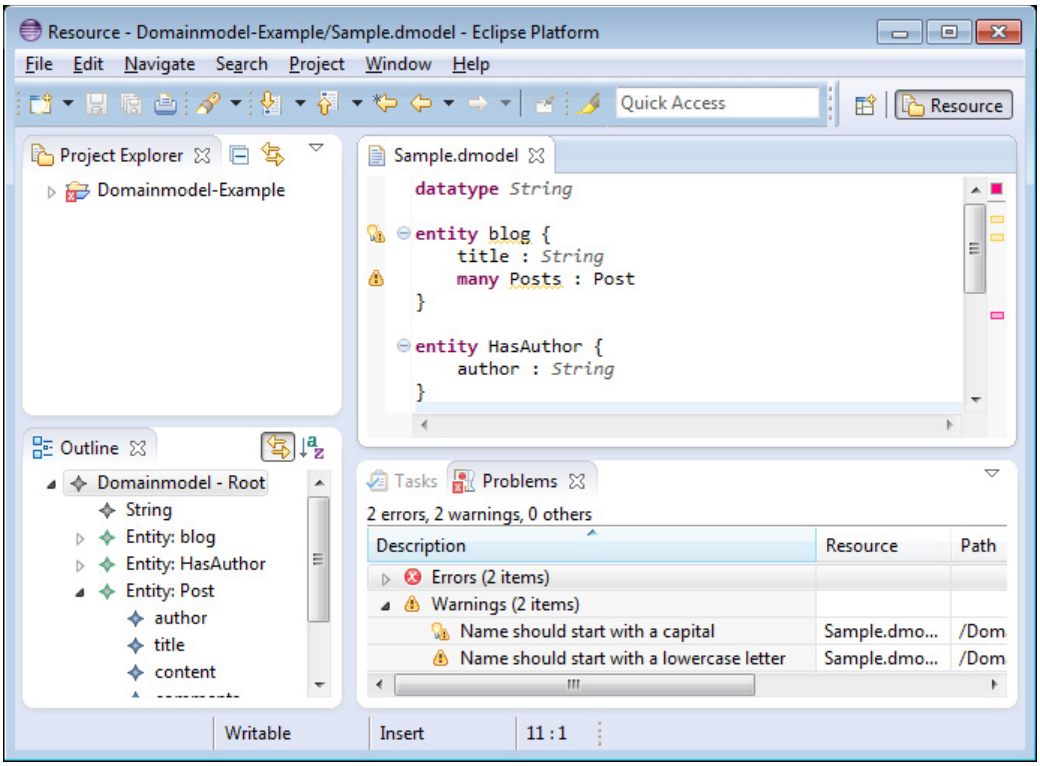

Abbildung 2-5: *Ein beispielhafter Editor für eine DSL*

[11] Das Xtext-Framework – http://www.eclipse.org/Xtext/

[12] Das openArchitectureWare-Projekt – http://www.openarchitectureware.org/

[13] Webpräsenz von itemis – http://www.itemis.de/

An diesem Beispiel ist bereits erkennbar, daß sich das Xtext-Framework sehr stark an Eclipse orientiert – die gesamte Editorplattform basiert auf dem entsprechenden Modell.

Xtext ist außerdem in der Lage, einen Codegenerator für eigene Sprachen zu erzeugen – das ist besonders im Hinblick auf die Erstellung externer DSLs ein wichtiger Aspekt, da solche Sprachen, wie im entsprechenden Abschnitt erwähnt, zurück in GPLs gewandelt werden müssen, um eine effektive und sinnvolle Nutzung zu ermöglichen.

2.4.1 Xtext-Grammatiken

Die Sprachgrammatik einer domainspezifischen Sprache beschreibt, analog zur Grammatik natürlicher Sprachen, grundsätzliche Regeln, nach denen ihre Konstrukte aufgebaut werden können. Auch in Xtext steht im Kern der Entwicklung einer DSL die Erstellung einer solchen Grammatik. Xtext bietet zu dessen Erstellung Java-nahe APIs, die den Entwickler bei dieser Aufgabe unterstützen sollen.

Die Beschreibung eines solchen Sprachgerüsts muß nach gewissen, aufeinander aufbauenden Regeln erfolgen – eine DSL-eigene Grammatik muß semantisch und syntaktisch der von Xtext vorgeschriebenen Definition – einer sogenannten Meta-Grammatik – entsprechen. Diese DSL-Grammatik ist schließlich dann Ausgangspunkt für Code, der in dieser neuen DSL geschrieben wird und dient diesem als Validator. Listing 2-6 beschreibt diesen Ablauf grob.

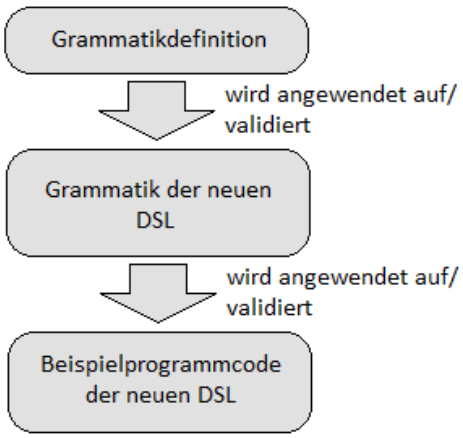

Abbildung 2-6: *Schematische Darstellung der Hierarchie einer Xtext-DSL*

Herauszuheben ist zusätzlich, daß Xtext eine Menge gemeinsamer „Terminal Rules“ unter der Grammatik `org.eclipse.xtext.common.Terminals` enthält, die der Entwickler einer eigenen DSL direkt verwenden kann – diese Regeln beschreiben primitive Datentypen wie Integer, Strings oder Kommentare, wie in Listing 2.8 dargestellt. Seit Xtext 2.0 existiert zudem die Basisgrammatik `org.eclipse.xtext.xbase.Xbase` zur einfachen Definition Java-naher Elemente wie beispielsweise Collections oder auch zur Beschreibung von Lambda-Ausdrücken.

```
grammar org.eclipse.xtext.common.Terminals
  hidden(WS, ML_COMMENT, SL_COMMENT)

import "http://www.eclipse.org/emf/2002/Ecore" as ecore

terminal ID :
  '^'?('a'..'z'|'A'..'Z'|'_')('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;
terminal INT returns ecore::EInt:
  ('0'..'9')+;
terminal STRING  :
  '"' ( '\\'('b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\') | !('\\'|'"') )* '"' |
  "'" ( '\\'('b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\') | !('\\'|"'") )* "'";
terminal ML_COMMENT  :
  '/*' -> '*/';
terminal SL_COMMENT :
  '//' !('\n'|'\r')* ('\r'? '\n')?;

terminal WS  :
  (' '|'\t'|'\r'|'\n')+;

terminal ANY_OTHER:
  .;
```

Abbildung 2-7: *Terminals-Grammatik für Xtext*

2.4.2 Funktionsweise

Die Grammatik wird nach ihrer Erstellung innerhalb der Eclipse-IDE als MWE[14]-Workflow ausgeführt. Hierbei wird aus der Grammatik heraus mit Hilfe des Parsergenerators ANTLR[15] ein Parser für die DSL erzeugt, der Modelle aus Texten, die der Definition der Grammatik entsprechen, generiert. Das Parsing kann in vier Phasen eingeteilt werden:

Lexing	Erzeugung von Token (Markern) aus Elementarregeln
Parsing	Kreation von Objekten aus Parserregeln, Erzeugung des AST (*abstract syntax tree*) aus diesen Objekten
Linking	Erstellung der Querverweise zwischen AST-Elementen
Validating	Überprüfung des erzeugten Parsermodells auf Validität

Tabelle 2-8: *Phasen der Parsererstellung mit ANTLR*

Weiterhin werden Java-Klassen – Artefakte genannt – für Teilaspekte des Editors der Sprache erzeugt, die unter anderem folgende Funktionalitäten umfassen:

- Syntax-Hervorhebung
- automatische Validierung
- Darstellung der Programmstruktur (Outline)
- Unit-Tests
- Code-Formatierung
- Quick-Fix-Vorschläge
- Code-Generierung

[14] Modeling Workflow Engine, ein Mittel zur Erzeugung von Projektstrukturen

[15] ANTLR, ein Parsergenerator für DSLs – http://www.antlr.org/

Diese erzeugten Klassen – Listing 2-9 bietet die Darstellung wesentlicher Teile eines Beispielprojekts sowie die Struktur eines Xtext-Projekts insgesamt – können vom Entwickler der DSL individuell angepaßt werden.

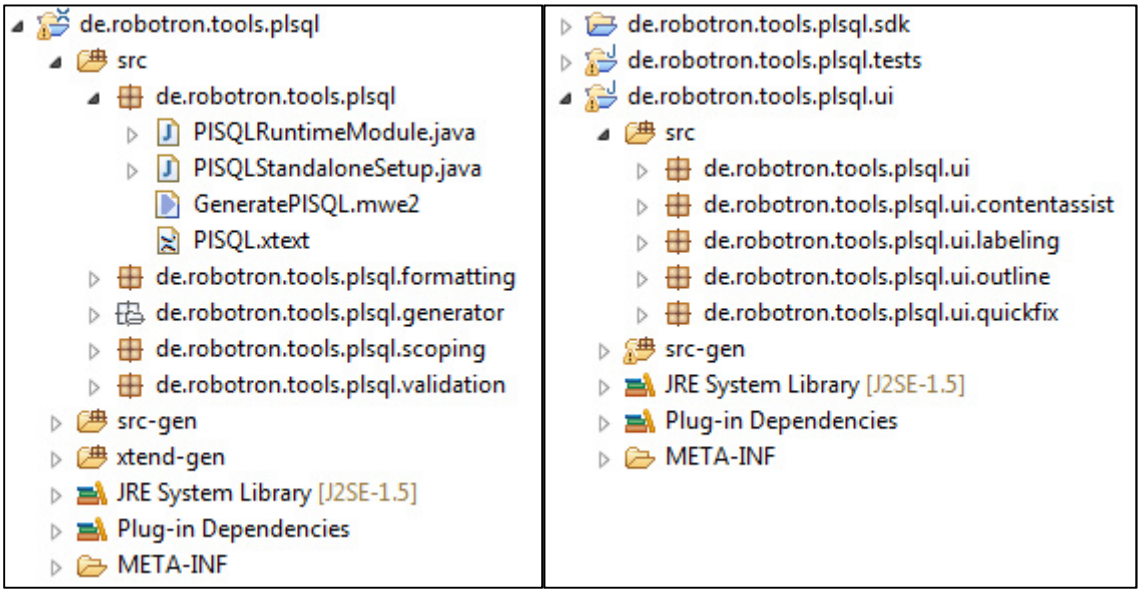

Abbildung 2-9: *Eine Xtext-Projektstruktur*

Das Xtext-Framework bietet für die Erzeugung dieser Sprachartefakte eine vorkonfigurierte Workflow-Datei, die Instruktionen dafür enthält, welche Fragmente für die Sprache erzeugt werden sollen – Listing 2-10 stellt einen Ausschnitt einer solchen Datei dar.

```
7  var grammarURI = "classpath:/de/robotron/example/domainmodel/Domainmodel.xtext"
8  var file.extensions = "dmodel"
9  var projectName = "de.robotron.example.domainmodel"
10 var runtimeProject = "../${projectName}"
11
12 Workflow {
13     bean = StandaloneSetup {
14             scanClassPath = true
15             platformUri = "${runtimeProject}/.."
16         }
17         component = DirectoryCleaner {
18             directory = "${runtimeProject}/src-gen"
19         }
20
```

Abbildung 2-10: *MWE-Workflow (Ausschnitt)*

Sowohl der erzeugte Parser als auch andere Compiler-Komponenten sind Eclipse-unabhängig und können in allen Java-Umgebungen genutzt werden. Die erstellte Sprache könnte zudem in einem reinen, von Eclipse unabhängigen Java-Umfeld bereitgestellt werden – eine Integration in Eclipse ist jedoch empfohlen, da das Framework als Gesamtes auf die IDE ausgerichtet ist.

2.5 Sonar

Sonar[16] (seit Mitte Juni 2013 SonarQube) ist ein Codeanalysetool, ausgerichtet auf die Untersuchung komplexer Projekte verschiedener Sprachen. Für das Werkzeug werden, teils kommerziell, Erweiterungen für verschiedene Programmiersprachen, u.a. Java, C, C# oder COBOL[17], unterstützt.

Das Tool bietet dem Nutzer die Möglichkeit, ein Softwareprojekt im Gesamten zu analysieren und auf eventuelle schlechte Praktiken, Sicherheitslücken oder Bug-Quellen hinzuweisen. Der Code wird hierbei an festgelegten Programmierungsregeln gemessen, das Güteergebnis ergibt sich aus dem Grade der Konformität mit diesen Regeln. Zusätzlich wird die Komplexität des Codes in seinen Einzelheiten, also am Beispiel von Java hierarchisch aufsteigend nach seinen Methoden, seinen Klassen und den Dateien im Gesamten, beurteilt. Zudem erhält der Nutzer bei der Analyse die Statistik über den Gesamtumfang des Codes (LOC) sowie prozentuale Angaben der Anteile des Codes, der sich wiederholt oder aus Kommentaren besteht.

Dem Entwickler wird das Ergebnis schließlich, wie in Listing 2-11 zu sehen, aufgeschlüsselt als Webseite präsentiert.

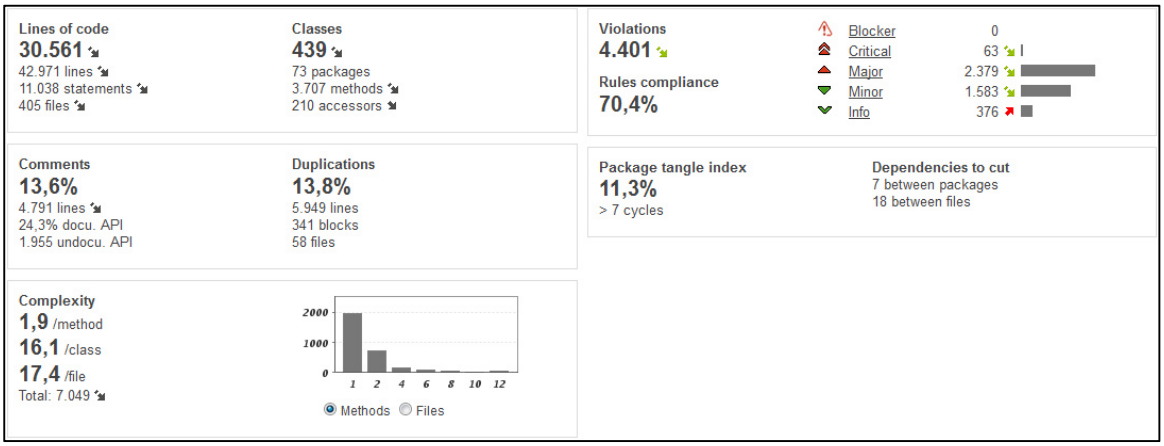

Abbildung 2-11: *Code-Statistiken des Sonar-Tools*

Auffälligkeiten in der Programmstruktur werden nach ihrer Kritikalität – ein Aspekt, der für jede Regelverletzung konkret bestimmt werden kann – sortiert und aufsummiert ausgegeben. Das Tool bietet nach erfolgter Analyse die Option, genauer auf die Fehlerarten einzugehen – hierbei wird dem Nutzer gezeigt, welche Verletzungen wie oft aufgetreten sind (Listing 2-12). Der Entwickler kann daraufhin direkt an die entsprechende Stelle im Projekt springen, um den Fehler zu korrigieren.

---

[16] Onlinepräsenz des Sonar-Tools – http://www.sonarsource.org/

[17] Common Business Oriented Language, eine in den 60er-Jahren entstandene Programmiersprache

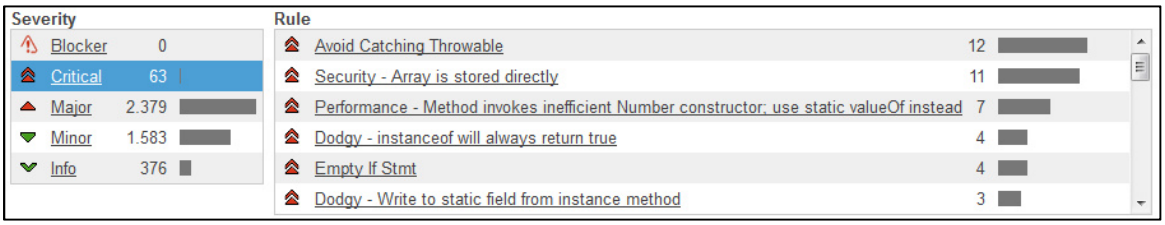

Abbildung 2-12: *Fehlerüberblick im Sonar-Tool (Ausschnitt)*

Für eine tiefergreifende Übersicht bietet das Werkzeug außerdem die Möglichkeit, so genannte „Hotspots“ – also Brennpunkte – zu betrachten. Diese Hotspots geben unter anderem Aufschluß über häufig verletzte Regeln sowie Klassen höherer Komplexität (Listing 2-13). Dieser Überblick vereinfacht es dem Entwickler, Ansatzpunkte für eine eventuelle Refaktorierung des Codes zu finden.

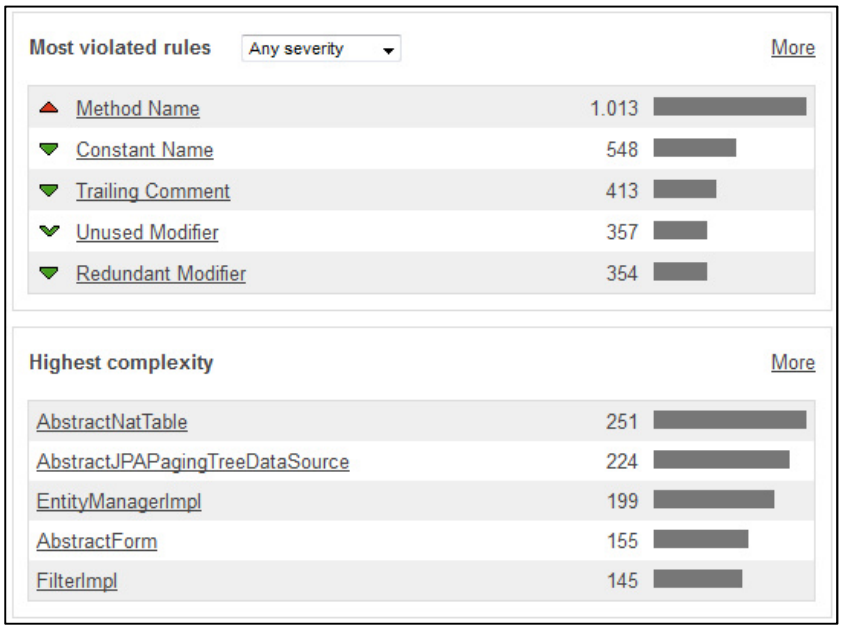

Abbildung 2-13: *Auflistung von Hotspots in Sonar (Ausschnitt)*

Sonar enthält für seine unterstützten Sprachen bereits eine Vielzahl an Regeln bezüglich verschiedener Aspekte wie Sicherheitsproblemen, möglicher Speicherlecks oder schlechtem Programmierstil – der Benutzer hat jedoch zusätzlich die Möglichkeit, für ein spezielles Produktumfeld nötige zusätzliche Richtlinien zu definieren und mit entsprechender Rangordnung zu versehen. Listing 2-14 zeigt beispielhaft einen Ausschnitt der existierenden Festlegungen für Java.

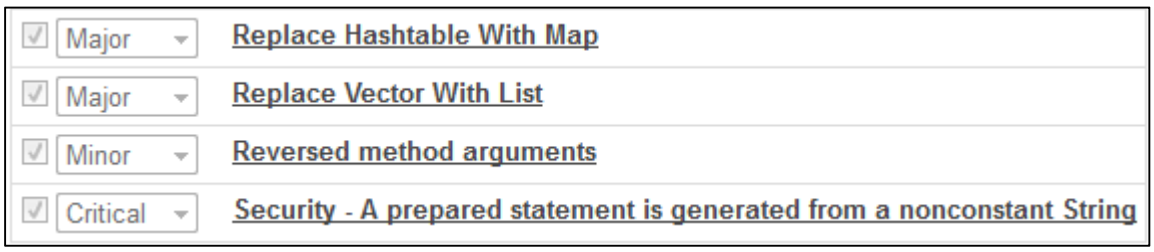

Abbildung 2-14: *Regeldefinitionen in Sonar (Ausschnitt)*

Die Analyse des Codes läuft neben der Entwicklung des Projektes ab – Sonar bietet im Bezug hierauf die Funktionalität, Informationen über die Konformität und den Umfang des Codes verlaufsmäßig über ein Zeitintervall zu betrachten (Listing 2-15). Tendenzen über die Richtung der Codeentwicklung können mit Hilfe dieser so genannten „Time Machine“ (dt. Zeitmaschine) so schnell erkannt und aufkommende Fehlerquellen zeitnah beseitigt werden.

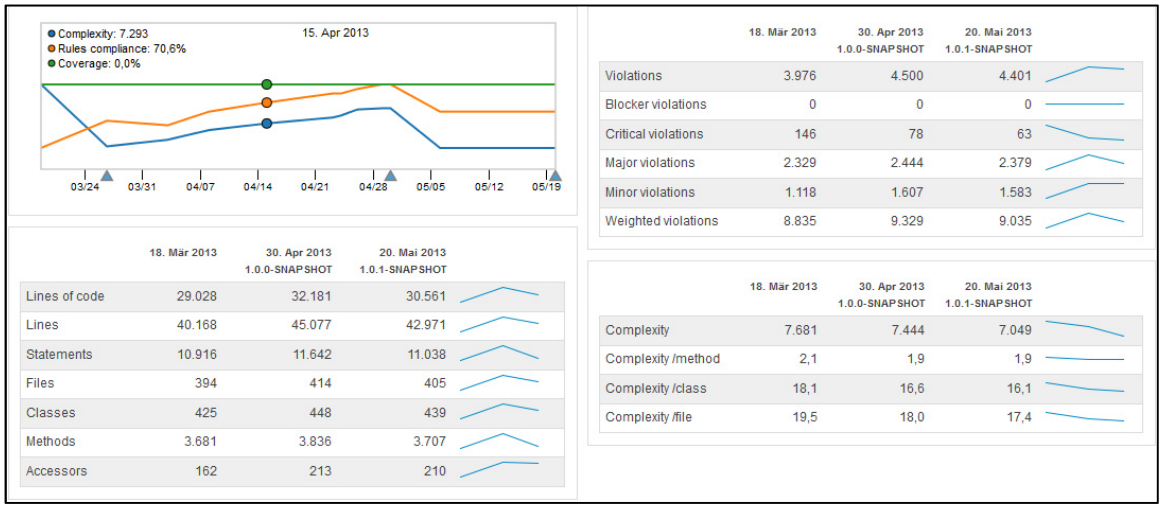

Abbildung 2-15: *Übersicht des Projektstatistik-Verlaufs im Sonar-Tool (Ausschnitt)*

Im Normalfall erfolgt die Analyse durch das zugehörige Tool „Sonar Runner“ – durch einen Konsolenbefehl wird das Tool zur Untersuchung eines speziellen Projekts angestoßen und daraufhin Analyseergebnisse generiert und präsentiert. Für eine nähere Integration mit dem Eclipse-Framework bietet Sonar außerdem ein spezielles Plugin, das es ermöglicht, die Untersuchung des Programmcodes kompletter Projekte in der IDE selbst durchzuführen.

3 Anforderungsanalyse und Konzept

In den kommenden Abschnitten erfolgt eine grundsätzliche und detaillierte Analyse der sich an die zu lösenden Aufgaben stellenden Anforderungen. Sie stützt sich wesentlich auf ein für die Konzeption erstelltes Pflichtenheft und legt spezielles Augenmerk auf die durch die Entwicklung der Software zu lösenden Probleme.

Zusätzlich werden Konzept und Struktur der letztendlichen Lösung in ihren Grundzügen dargestellt.

3.1 Aufgabenbereiche

Während des Verschaffens eines Überblicks über die Gesamtstruktur und -komplexität der Teilaufgaben kristallisierten sich grundsätzlich drei Aufgabenbereiche heraus, in denen die zu entwickelnde Software einen Nutzen finden soll – die gesteigerte Möglichkeit der Codeanalyse, die verbesserte Codewartung sowie die generelle Codeverbesserung. Diese Bereiche überschneiden sich zum Teil untereinander, sodaß zu ihrer Unterscheidung lediglich eine grobe Abgrenzung möglich ist.

Unter gesteigerter **Codeanalyse** als dem ersten der Aufgabenbereiche ist zu verstehen, daß sowohl dem Softwareentwickler als auch eventuellen Projektleitern sowie Testern eine einfachere Möglichkeit geboten wird, den Code eines umfangreichen Gesamtprojekts zu überblicken und bei zunehmenden Komplexitäts- oder Programmstilproblemen eine Code-Refaktorierung in Betracht zu ziehen. Eine verbesserte Analyse des Gesamtcodes ebnet zusätzlich den Weg zur Erreichung der Ziele, die durch die anderen Aufgabenbereiche vorgegeben sind – verbesserter Codewartung und grundlegender Verbesserung der Codequalität.

Der Aspekt der **Codewartung** umfaßt im Speziellen den Anwendungsfall der im vorigen Absatz angesprochenen Refaktorierungen. Dem Entwickler sollen durch die zu erstellende Software konkrete Ansatzpunkte geliefert werden, für die eine Wartung und Neustrukturierung notwendig und sinnvoll ist. Dies beschränkt sich nicht nur auf *bad practices*[18], sondern auch auf zu stark angewachsene Komplexität in einzelnen Teilen des Gesamtprojekts – beispielsweise zu stark verzweigte Auswahlen, die im Zuge verschiedener Änderungen am Projekt immer wieder erweitert werden mußten, jedoch nie korrekt modularisiert und aufgeteilt worden sind.

[18] „bad practice“ – schlechte Programmierpraktiken, schlechter Programmierstil

Codeverbesserung schließlich bezieht sich darauf, den Entwickler von vornherein auf schlechte Praktiken hinzuweisen, um Bugs vorzubeugen und spätere Überarbeitungen des Codes seltener nötig zu machen – damit wird letztendlich auch die Qualität des Programmcodes wesentlich gesteigert. Die Erfüllung dieser Anforderung kann von der Software lediglich unterstützend erfolgen, d.h. die schließliche Lösung kann den Entwickler lediglich zu besserem Programmierstil anleiten, ihn jedoch in keinem Falle dazu zwingen, vorgeschlagene optimalere Praktiken zu nutzen.

Bei der Betrachtung der Anforderungen war stets zwischen zwei Teilprodukten – einerseits der Erstellung einer Sprachgrammatik für eine geringfügig angepaßte PL/SQL-Sprache und andererseits der Implementierung eines Plugins für das Codeanalysetool Sonar – zu unterscheiden. Diese beiden Teillösungen beziehen sich – wie in den beiden Folgeabschnitten genauer erläutert – auf verschiedene Spektren des Gesamtaufgabenbereiches.

3.2 Anforderungen an die Zielgrammatik

Die zu erstellende PL/SQL-Grammatik als erste Etappe der Lösung des Gesamtproblems betrifft im Wesentlichen die beschriebenen Aspekte der Codewartung und -verbesserung. Durch die Grammatik soll gewährleistet sein, daß dem Entwickler zur Programmierzeit Hinweise und Fehler im Code angezeigt werden sowie gegebenenfalls Lösungsvorschläge angeboten werden. Zusätzlich soll auch bestehender Code durch die Grammatik validiert werden können, um dort ebenfalls Warnungen und Programmfehler zu markieren.

Im Speziellen stellen sich an die Grammatik zwei Anforderungen – einerseits ist das der Anspruch an den Umfang derselben, andererseits an ihre Erweiterbarkeit.

Der **Umfang** der letztendlich erstellten Grammatik muß alle wesentlichen Teile der PL/SQL-Sprache abbilden können – insbesondere wird jedoch nicht der Anspruch gestellt, alle Konstrukte und ihre Möglichkeiten abzudecken. Eine solche komplette Implementierung besäße als Erstversion eine in vielfacher Hinsicht zu hohe Komplexität, verbunden mit sehr hohem Zeitaufwand für ihre Erstellung. Die Grammatik soll jedoch den Großteil aller verwendbaren Konstrukte zumindest in ihren Grundformen kennen, häufig verwendete Varianten sollen ausführlicher definiert sein. Insbesondere hinsichtlich der Entwicklung für ein konkretes Unternehmen besteht hierfür die Anforderung, daß alle in diesem Umfeld genutzten Sprachelemente bekannt und validierbar sind.

Die aus der Grammatik heraus stattfindende Validierung soll lediglich der Notwendigkeitsbedingung genügen, darüber hinaus jedoch nicht hinreichend sein. Das bedeutet, daß jeder im Rahmen der eingeschränkten Grammatik korrekte und nach Sprachspezifikation erlaubte PL/SQL-Programmausschnitt von der Grammatik so gelesen werden können muß, daß zur Validierung keine Fehler fälschlicherweise markiert werden. Dies bezieht sich jedoch nicht auf Warnungen bezüglich schlechten Stils, die auch in korrektem Code auftauchen können. Der explizite Ausschluß dessen, daß die erstellte Grammatik hinrei-

chend bezüglich der Validierung ist, bedeutet, daß nicht jedes fehlerhafte Sprachkonstrukt in jedem Falle als Fehler markiert werden muß. Es besteht in der zu diesem Zwecke erstellten Grammatik beispielsweise die Möglichkeit, daß implizite Regeln nicht korrekt validiert werden und nicht als Programmfehler oder schlechte Praktiken auffallen – es ist zu erreichen, daß ein überwiegend großer Anteil aller Grammatikfehler und Warnungen im Programmcode zu erkennen und zu markieren sind.

Aus dieser Spezifikation des Grammatikumfangs ergibt sich die Notwendigkeit der Anforderung der **Erweiterbarkeit**. Die im Zuge dieser Arbeit erstellte PL/SQL-Grammatik soll kein Definitivum darstellen – Entwicklern soll die Möglichkeit gegeben werden, zusätzliche Sprachkonstrukte zu ergänzen sowie in Richtung einer hinreichenden Grammatik arbeiten zu können. Diese Anforderung begründet sich zudem aus der Tatsache heraus, daß die Sprache selbst noch im Fluß ist und seitens Oracle zukünftige Änderungen nicht auszuschließen sind.

3.3 Anforderungen an die Integration in Sonar

Die Integration der Grammatik unter dem Gesichtspunkt besserer Codeanalyse mit Hilfe des Sonar-Tools als zweiter Entwicklungsschritt geht in erster Form mit beschränkten Anforderungen einher. Dies gründet sich darauf, daß die Implementierung in das Tool lediglich beispielhaft erfolgen soll, das Plugin also keine vollständige Integration bieten soll, um dem interessierten Unternehmen – im konkreten Falle Robotron – die Möglichkeit zu geben, den Nutzen des Tools selbst besser einzuschätzen und darauf aufbauend für oder gegen eine Erweiterung in Zukunft zu entscheiden.

Das Plugin ist daher lediglich mit Minimalumfang zu entwickeln – es soll einen Überblick über den Codeumfang bieten und eine kleine Auswahl an gesetzten Codekonformitätsregeln untersuchen. Dies umfaßt jedoch, nativ von der Sonar-Umgebung unterstützt, zusätzlich weitere Funktionalitäten wie der Möglichkeit des *drill down*[19] – das bedeutet der Nachverfolgung auftretender Konformitätsverletzungen bis an konkrete Stellen im Code und ihrer Behebung – oder der Erstellung zeitlicher Verläufe.

Aus diesen Darstellungen heraus ergibt sich jedoch auch hier die Anforderung der **Erweiterbarkeit** – die entwickelte Software darf wie zuvor die Grammatik nicht final sein und erhebt daher den Anspruch, für Folgeentwickler gut strukturiert sowie dokumentiert zu sein.

Eine weitere natürliche Anforderung an das Plugin ist die **Korrektheit** der erzeugten Statistiken, d.h. daß die Analysen, die im Prototypen der Software durchgeführt werden, in jedem Falle exakt sind, also keine falschen Werte liefern.

---

[19] „drill down“ – wörtlich: Hinabbohren durch die Programmstruktur, d.h. konkrete Fehlersuche

3.4 Anforderungen hinsichtlich der zu erreichenden Ziele

Die folgenden Unterabschnitte gehen entsprechend der Aufgabenstellung konkret auf die Anforderungen bezüglich der Erreichung der beiden Hauptziele – Wartbarkeit und Analyseverbesserung – ein. Es werden hierbei konkret Ist- und Soll-Zustände verglichen und die Verbesserungen, die die Entwicklung der Grammatik als auch des Plugins mit sich bringen, einer detaillierten Untersuchung unterworfen.

3.4.1 Wartbarkeit

Es existieren verschiedene Editoren, die die Programmiersprache PL/SQL fehlerfrei parsen können – im vorherigen Kapitel wurde der PL/SQL Developer von Oracle selbst erwähnt – diese bestehende Software läßt sich jedoch nicht auf spezielle Umstände zuschneiden. Als prominentester Anwendungsfall zur Verdeutlichung des Nutzens einer eigens kreierten Grammatik für eine solche modifizierte DSL ist die Möglichkeit zu nennen, firmeninterne Konventionen in die Grammatik direkt einzuarbeiten. Konkretes Beispiel wäre hierfür die Anweisung „Paketnamen müssen mit dem Präfix *pkg_* beginnen“ – bezüglich der Wartbarkeit als zu erreichendem Ziel besteht folglich also die Anforderung, derlei spezielle Bedingungen mit zu erfassen und die Funktionalität der Software zu integrieren.

Neben dieser über die Funktion üblicher Editoren hinausgehenden Anforderung fordert dieses Ziel zudem die in den vorigen Abschnitten beschriebenen grundlegenden Anforderungen bezüglich der Korrektheit – um ein sinnvolles Maß an Wartbarkeit zu erreichen, muß mit dem Produkt die Möglichkeit gegeben sein, fast jeglichen PL/SQL-Code korrekt zu parsen und zumindest den Großteil aller Fehler zu erkennen. Es soll jedoch noch keinerlei Prüfung zwischen Querverweisen stattfinden, beispielsweise hinsichtlich der Frage, ob bestimmte Variablen bereits deklariert worden sind, bevor sie genutzt werden.

Zusätzlich ergibt sich die optionale Anforderung, Fehler in den konkreten Stellen des Programmcodes zu markieren und sie direkt und unkompliziert editieren zu können.

3.4.2 Analyseverbesserung

Das Ziel der Analyseverbesserung ist prinzipiell in zwei verschiedene Aspekte zu teilen – es ist hier einerseits die Verbesserung der qualitativen Analyse sowie andererseits die Kreation einer Möglichkeit einer quantitativen Analyse zu betrachten.

Als qualitativer Aspekt ist neben syntaktischer Codekorrektheit auch die Verwendung schlechter Codepraktiken zu betrachten. Untersuchungen von Softwareprojekten bezüglich dieser Elemente sind in existierenden Tools und Umgebungen nur zum Teil gegeben – insbesondere hinsichtlich *bad practices* sowie häufiger Ursachen für Bugs kann eine Editorsoftware für eine Sprache im Normalfall keinerlei Unterstützung bieten. Für die zu

erstellende Software ergibt sich also hieraus die Anforderung, Statistiken über die Korrektheit des Codes sowohl im Gesamten als auch im Einzelnen erstellen zu können. Fehler sollen nach Kritikalität – also unterschieden nach schwerwiegenden Fehlern und Informationen/Hinweisen – und Art geordnet aufsummiert werden und in zeitlichen Verläufen hinsichtlich ihrer Häufigkeit betrachtet werden können.

Quantitative Aspekte beschreiben im Gegensatz hierzu die makroskopische Sicht auf den Programmcode – für die zu erstellende Software ergibt sich hinsichtlich der Analyseverbesserung die Anforderung, Statistiken darüber zu erzeugen, welchen Umfang der Code hat – das heißt also unter anderem, wie viele Codezeilen und Packages er umfaßt und wie komplex die einzelnen Teile und der Code im Gesamten ist. In existierender Software findet sich eine solche Möglichkeit nicht – insbesondere bezüglich der Komplexität können ohne tiefgehende manuelle Untersuchungen in den seltensten Fällen genaue Aussagen getroffen werden, lediglich Benchmarks[20] bieten in dieser Hinsicht Anhaltspunkte. Analog zur qualitativen Analyse soll auch für die Untersuchung der Quantität ein zeitlicher Verlauf bereitgestellt werden, um beispielsweise einem zu starken Komplexitätswachstum zeitig entgegenwirken zu können.

3.5 Alternativen zur Zielerreichung

Um einen Überblick über die grundsätzlichen Möglichkeiten zu erhalten, die sich bieten, um die gesetzten Ziele zu erreichen, werden im Zuge der Anforderungsanalyse verschiedene Tools und ihre Möglichkeiten analysiert.

Hinsichtlich des Ziels der verbesserten **Wartbarkeit** des Programmcodes war von Beginn an sehr klar, daß eine eigene Grammatik für die angepaßte PL/SQL-Sprache zu schreiben ist, die schließlich dazu dienen soll, Programmfehler zu erkennen und den Entwickler bei ihrer Beseitigung zu unterstützen. Hierfür bieten sich im Wesentlichen zwei Möglichkeiten – die Grammatik kann unter dem Tool ANTLRWorks[21] oder mit Hilfe des Xtext-Frameworks kreiert werden. Da das Xtext-Framework direkt auf ANTLR aufsetzt und für die Erstellung von Grammatiken zusätzliche Hilfen im Rahmen der Eclipse IDE bietet, wird die Grammatikerstellung durch dieses Tool präferiert.

Weiterhin muß entschieden werden, in welchen Teil des Endprodukts sich die Codewartung und -validierung schließlich überwiegend verlagern soll. Hierfür existieren auf Basis voriger Entscheidungen drei Optionen – die Validierung kann fest in der Grammatik beschrieben sein, sie kann im durch das Xtext-Framework für die DSL erzeugten Editor selbst stattfinden oder sie kann in der Integration im Sonar-Tool erfolgen. Bevorzugt wird

[20] Benchmark – (oftmals zeitliche) Geschwindigkeitsmessung von Programmen und Programmteilen

[21] ANTLRWorks, ein Programmierumfeld zur Grammatikerstellung – http://www.antlr3.org/works/

bezüglich dieses Aspekts eine Mischung aus der ersten und der letzten Option – eine feste Beschreibung sämtlicher genutzter Grundkonstrukte der Sprache in der Grammatik selbst gepaart mit der Validierung firmeninterner Spezifikationen über das Sonar-Tool. Diese Varianten werden bezüglich der Flexibilität der Änderungen in erlaubten und korrekten Sprachelementen bevorzugt – da eine Änderung der Grammatik oftmals eine komplette Neukompilierung der auf sie aufbauenden Projekte bedeutet, muß Wert darauf gelegt werden, daß sie sich auf die Elemente beschränkt, die von Oracle festgeschrieben sind und die sich sehr wahrscheinlich nicht ändern. Firmenspezifische Elemente und *bad practices* jedoch können im Fluß sein und sich kurzfristig ändern – deshalb sollten diese in der Sonar-Umgebung selbst als Konformitätsregeln beschrieben werden, die sich ohne großen Aufwand ändern lassen.

Bezüglich des Erreichens der **Analyseverbesserung** gilt als vorausgesetzt, daß das Sonar-Tool zum Einsatz kommen wird. Dies ist darin begründet, daß das Tool in der Unternehmung bereits in Verwendung ist und nun lediglich auf andere Sprachen ausgeweitet werden soll. Eine weitere Option hätte in der Möglichkeit bestanden, als Erweiterung für den aus der Xtext-Grammatik erstellten Editor in Eclipse ein eigenes Analysetool zu schreiben, welches passende Ergebnisse liefert. Auch aufgrund des sehr hohen Aufwands dieser Lösung wird diese Möglichkeit jedoch nicht näher in Betracht gezogen. Hierbei wird Wert auf eine Analyselösung im Web-Interface des Sonar-Tools gelegt – die Kreation eines Plugins mit Hilfe der Eclipse-Integrationsmöglichkeiten Sonars wurde erwogen, jedoch ebenfalls aus dem Gesichtspunkt dessen, daß existierende Projekte ins Web-Interface integriert sind und sich nur unter größerem Aufwand komplett in eine Eclipse-Umgebung einbetten lassen würden, verworfen.

Letztendlich bieten sich für das Sonar-Plugin selbst noch verschiedene Optionen - einerseits die Möglichkeit, es komplett neu zu kreieren, andererseits die Variante, das bestehende kommerzielle PL/SQL-Plugin zu erwerben und darauf aufbauende Anpassungen und Änderungen bezüglich firmenspezifischer Regeln zu beschreiben. Um die Flexibilität in der Entwicklung – im Speziellen hinsichtlich der Freiheiten bei der Softwareverwendung durch gegebene Lizenzen – zu gewährleisten und auf spezielle Unternehmsbedürfnisse und -wünsche konkret Bezug nehmen zu können, wird in diesem Aspekt die Erstellung eines komplett neues PL/SQL-Plugins für die Sonarumgebung bevorzugt. Es soll lediglich in seinen Grundzügen auf existierenden Plugins für andere Sprachen, insbesondere bezüglich Aufbau und Darstellung der Statistiken, basieren.

3.6 Konzept

An dieser Stelle sollen die Konzeptionen zur Lösung der gestellten Probleme – das bedeutet in diesem Falle die Planung und Architektur der zu erstellenden Software - dargestellt werden. Hierbei wird wie zuvor Wert auf eine Zweiteilung der beiden Teilprobleme – der PL/SQL-Grammatik und des Sonar-Plugins – gelegt.

3.6.1 PL/SQL-Grammatik

Die Konzeption für die PL/SQL-Grammatik ist in zweierlei Abschnitte zu unterscheiden; einerseits müssen hinsichtlich der Struktur der Grammatik Entscheidungen getroffen werden, andererseits ist es nötig, einen konkreten Pool an zu erstellenden Funktionen zu erarbeiten.

Die Modellierung der Grammatik erfolgt aufbauend auf dieses Wissen nach einer auf drei Eckpfeilern fußenden Philosophie – das sind einerseits der hierarchische Aufbau der Grammatik, zweitens ihre logische Struktur sowie zuletzt ein auf Unternehmsansprüche zugeschnittener Funktionsumfang.

Die Erstellung der Grammatik erfolgt iterativ – das heißt, daß zuerst eine Grundversion erstellt wird. Diese wird schließlich an produktiv genutztem und gesichert korrektem Code validiert, woraufhin eine weitere, verbesserte Version der Grammatik erarbeitet wird und sich der Zyklus wiederholt.

3.6.1.1 Aufbau

Wie zuvor beschrieben hat die Grammatik einen Anspruch auf Verständlichkeit und Erweiterbarkeit. Hierfür ist es nötig, daß ein mit ihr nicht vertrauter Bearbeiter sich in kurzer Zeit einen Überblick verschaffen kann. Ein solcher Anspruch – neben den Grundregeln guten Programmierstils – ist bei der Ausarbeitung des Konzepts stets zu beachten.

Hinsichtlich dessen spielt bei der Erstellung der Grammatik der eingangs erwähnte **hierarchische Aufbau** eine große Rolle. Die Grammatik ist als Baum zu betrachten – wobei die Wurzel das Kernelement, also im konkreten Fall eine komplette Programmeinheit, darstellt und die Äste sich auf spezielle Funktionen und Hilfsfunktionen beziehen.

Aus dieser Metapher heraus läßt sich die Planung der Hierarchisierung gut darstellen – Listing 3-1 soll hierfür einen bildhaften Überblick verschaffen. Ausschnitthaft ist die Lage des `INSERT`-Statements und seiner `WHERE`-Klausel gekennzeichnet. Hierbei ist anzumerken, daß die `WHERE`-Klausel auch in anderen Teilen der Gesamtsprache verwendet wird – beispielsweise im PL/SQL-Bereich. Grammatikalische Elemente, die vorher definierte Klauseln ebenfalls beinhalten, sollen jedoch ebenfalls direkt auf sie zugreifen, Mehrfachdefinitionen sollen vermieden werden. Diese Gegebenheit verdeutlicht, daß die Darstellung als Baum lediglich als Vereinfachung dienen kann – tatsächlich vereinen sich die Gabelungen in tieferliegenden Ebenen in einigen Fällen erneut.

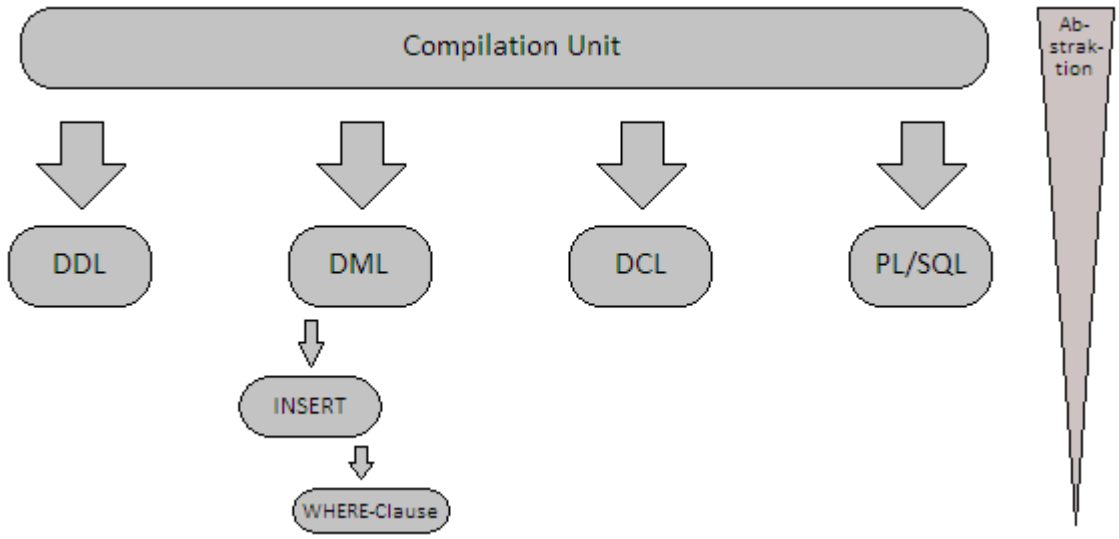

Abbildung 3-1: *Hierarchiestruktur der PL/SQL-Grammatik (Ausschnitt)*

Wie aus dem Bild zu erkennen ist, nimmt der Baum von oben nach unten an Abstraktion ab. Nahe der Wurzel sollen sich die abstraktesten Elemente befinden, diese sollen sich entsprechend in konkretere Einteilungen verzweigen und schließlich in den Blättern zu einzelnen Funktionen führen. Wie bereits zu erkennen ist, spielt die in den Grundlagen beschriebene Aufteilung der PL/SQL-Sprache eine wesentliche Rolle in der Hierarchisierung – „Compilation Units“, also komplette Programmblöcke, bestehen aus einzelnen SQL oder PL/SQL-Statements, die in DDL-, DML- sowie DCL-Statements unterteilt sind. Diese Aufteilung ist ein einfacher Weg, die Übersichtlichkeit der Gesamtgrammatik für den Entwickler sowie für externe Betrachter zu erhöhen.

Der zweite wichtige Aspekt bezüglich des formalen Grammatikaufbaus ist seine **logische Struktur**. Die einzelnen Ebenen des vorher erwähnten Baumes sollen nach ihrem Grad der Abstraktion geordnet nacheinander im Code auftauchen – dies vereinfacht das Finden einzelner Elemente und bietet eine gute grundlegende Infrastruktur für eventuelle Erweiterungen. Listing 3-2 zeigt die prinzipielle Konzeption des logischen Aufbaus.

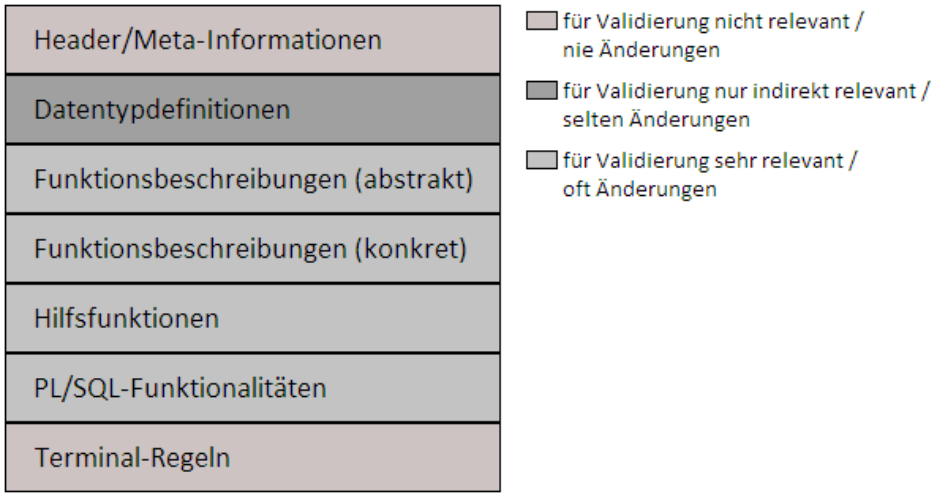

Abbildung 3-2: *Logischer Aufbau der PL/SQL-Grammatik*

Die Grammatik eröffnet aus programmtechnischen Gründen mit einigen Headerinformationen – anschließend werden alle Datentypen aufgelistet, die für sie von Belang sind; eine solche Clusterung ist wiederum sinnvoll für die Erweiterung und Anpassung, so unter anderem für den Fall, daß neue Datentypen nötig sind oder eine Modifizierung nötig wird. Dem folgend werden schließlich die abstrakten Funktionen gelistet – an dieser Stelle findet die Unterscheidung in die SQL-Teilsprachen statt. Als ein nächster Schritt werden dann alle einzelnen Funktionen nacheinander definiert – diese Funktionen besitzen in fast allen Fällen Helferfunktionen, die sich im nächsten Grammatikabschnitt befinden. Schließlich folgt der vom Rest geringfügig abgetrennte PL/SQL-Teil der Grammatik; die programmatischen und prozeduralen Aspekte der Spracherweiterung machen eine Trennung von der allgemeinen SQL-Sprache sinnvoll. Die Grammatik schließt mit angepaßten Terminal-Regeln – diese Konstrukte werden aus Konventionsgründen und der Tatsache geschuldet, daß sie nicht editiert werden sollen, an das Ende der Grammatik gesetzt.

Zur Sicherung der Qualität und Nutzbarkeit der Grammatik ist es weiterhin notwendig, Mehrdeutigkeiten und Redundanzen zu vermeiden – eine Nichtbeachtung dessen kann zu unerwartetem Validierungsverhalten führen und einem Entwickler, der die Grammatik erweitern möchte, zukünftig vor kaum auffindbare Probleme stellen.

Neben diesen beiden Basisphilosophien sowie den Qualitätssicherungsaspekten umfaßt das Konzept für die Grammatik natürlich auch eine gewisse Konformität bezüglich der Regeln der Programmierung. Abschnitte in der Grammatik selbst werden ausreichend gekennzeichnet, Bezeichner und Variablen werden aussagekräftig gewählt und der Code im Gesamten wird in angemessenem Maße kommentiert.

3.6.1.2 Funktionsumfang

Wie in der Anforderungsanalyse umrissen, soll die Grammatik keine komplette Abbildung der SQL-Sprache sein – sie soll sich lediglich auf **unternehmensrelevante Funktionen** konzentrieren und ihr Augenmerk auf die PL/SQL-Erweiterung legen. Das bedeutet, daß nur eine geringe Auswahl aller SQL-Funktionen von der Grammatik erfaßt wird sowie an einigen Stellen der Funktionen Vereinfachungen und Einschränkungen vorgenommen werden. Diese Eingrenzung ergibt sich aus dem enormen Umfang der Sprache – eine weitergehend komplette Implementierung wäre vom Umfang und der zur Verfügung stehenden Zeit her nicht machbar.

Listing 3-3 gibt einen groben Überblick über die Funktionen, die zu implementieren sind. Bei der Erstellung wird größter Wert auf PL/SQL-Elemente und -Typen (z.B. Cursors, Trigger, Packages) gelegt – es werden so gut wie alle Aspekte der Spracherweiterung implementiert. Im Gegensatz dazu werden viele der Funktionen der Grundsprache SQL nicht berührt – Beispiele für explizit nicht eingebaute Funktionen sind an dieser Stelle verschiedene `CREATE`- und `ALTER`-Statements wie `CREATE/ALTER DATABASE` sowie `DROP`-Statements. Lediglich die für die Nutzung in Verbindung mit der Erweiterung relevanten und die am meisten benutzten werden von der Grammatik erfaßt.

DDL	`CREATE PACKAGE, CREATE PACKAGE BODY, CREATE SEQUENCE, CREATE SYNONYM, CREATE TABLE, CREATE TRIGGER, CREATE TYPE, CREATE VIEW, ALTER TABLE, COMMENT, GRANT`
DML	`INSERT, DELETE, MERGE, UPDATE, SELECT`
DCL	`COMMIT, ROLLBACK, SAVEPOINT`
PL/SQL-Elemente	Block-Statements (`BEGIN … END`), programmatisch genutzte Funktionen, u.a.: • Funktionsaufrufe • SQL-Erweiterungen `SELECT INTO` • Zuweisungen `:=` • Schleifen `FOR, FORALL, WHILE` • Flußkontrolle `IF, CASE, GOTO, CONTINUE` • Prozedurkontrolle `OPEN, CLOSE, RETURN, EXIT, FETCH, RAISE` • Spezialfälle `NULL`

Tabelle 3-3: *Liste der in der Grammatik zu implementierenden Funktionen*

Bei der Erstellung der Grammatik ist sich so nahe wie möglich an die Sprachreferenz der SQL- sowie PL/SQL-Sprache[22] zu halten – die dort für Einzelfunktionen beschriebenen Flußdiagramme sind für die implementierten Funktionen möglichst vollständig abzubilden.

3.6.2 Sonar-Plugin

Ähnlich wie der zu erstellenden PL/SQL-Grammatik ist für die Integration eines PL/SQL-Sonar-Plugins im Vorfeld ein Konzept bezüglich seiner Struktur sowie seines Umfangs zu erarbeiten.

Das Konzept des Plugins ist im Wesentlichen darauf ausgelegt, ein *proof of concept*[23] für die Möglichkeit der Integration solch einer Sprache zu entwickeln sowie eine Infrastruktur für eventuelle Erweiterungen zu kreieren. Insbesondere ist anzumerken, daß zwischen der erstellten DSL-Grammatik und dem Sonar-Plugin in der zu implementierenden Version eine sehr lockere Bindung herrscht – die Grammatik dient an dieser Stelle lediglich als Referenz für die Erstellung einer stark vereinfachten Abbildung auf Basis des Squid-Tools[24] der Entwickler von Sonar. Diese konzeptionelle Entscheidung wurde grundsätzlich auf der Basis dessen getroffen, daß eine komplette Integration der Grammatik bezüglich Aufwand und Umfang um ein Vielfaches zu umfangreich wären.

---

[22] Oracle Database (PL/)SQL Language Reference – entsprechende Links finden sich in den Quellen

[23] *proof of concept* – sinngemäß: Beweis der Durchführbarkeit einer Aufgabe

[24] Sonar Squid – Ein Tool zur Erstellung eigener Metriken und Grammatiken im Sonar-Ökosystem

Analog zu bisher existierenden Sonar-Plugins erfolgt die Implementierung unter Beachtung der LGPL[25].

3.6.2.1 Aufbau

Im Allgemeinen geschieht die Entwicklung des Sonar-Plugins – wie auch bei Produkten für andere unterstützte Sprachen – mit Hilfe des Maven-Tools[26], als Plugin in die Eclipse-Umgebung integriert.

Der logische Aufbau des Plugins erfolgt nach dem Vorbild der existierenden Python-Integration für die Sonar-Umgebung[27]. Das bedeutet, daß das Gesamtprojekt in drei Teile zu unterscheiden ist (Listing 3-4) – dem **Plugin-Projekt** selbst, welches Funktionalitäten bezüglich der Schnittstelle zu Sonar beherbergt, dem **Squid-Projekt** des Plugins, in welchem Details zur verwendeten Grammatik, so unter anderem Parser und Lexer, beschrieben sind und schließlich dem **Checks-Projekt**, welches alle durchzuführenden Prüfungen bezüglich der Regelkonformitäten erfaßt.

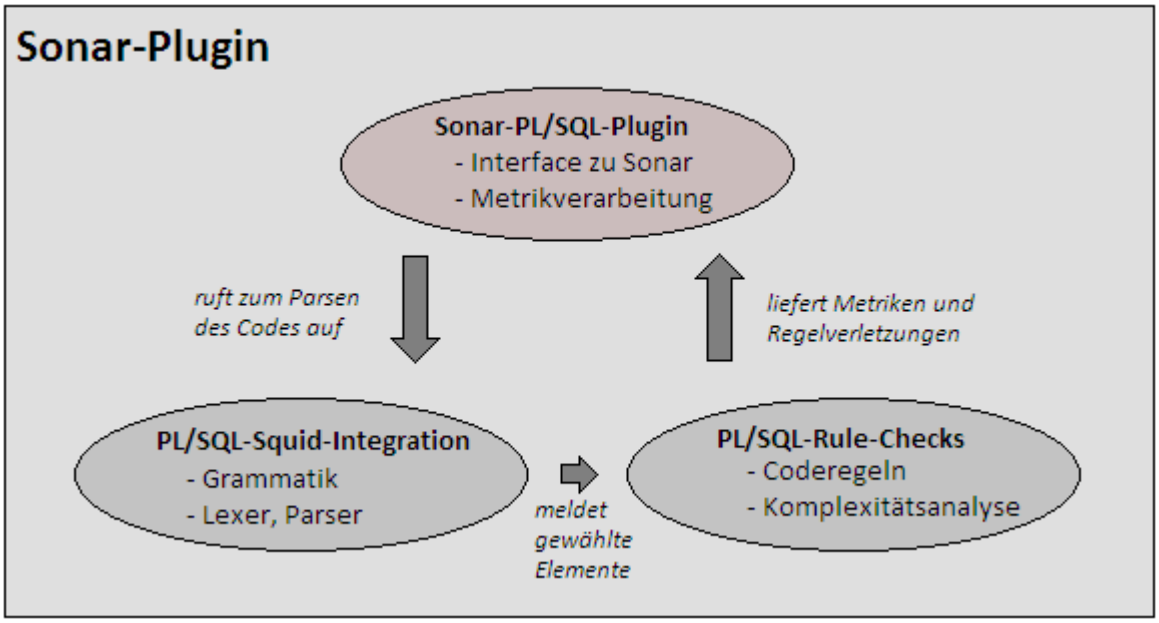

Abbildung 3-4: *Strukturkonzept des Sonar-PL/SQL-Plugins*

Die Implementierung des Projekts ist streng an diesem Modell ausgerichtet, um eine Übersichtlichkeit und Vergleichbarkeit und damit verbunden auch Wartbarkeit des Plugins in gewissem Maße zu gewährleisten.

---

[25] GNU LGPL, eine Softwarelizenz – http://www.gnu.org/licenses/lgpl-3.0.txt

[26] Maven, ein Build-Management-Tool der Apache Foundation – http://maven.apache.org/

[27] Informationen zum existierenden Python-Plugin – http://docs.codehaus.org/display/SONAR/Python+Plugin

3.6.2.2 Analyseumfang

Der Umfang des Plugins ist bereits konzeptuell sehr schmal gehalten, um lediglich eine einfache Darstellung dessen zu geben, welche Analysen das Sonar-Tool bezüglich des verwendeten Programmcodes durchzuführen vermag.

Bezüglich durchzuführender Tests sind folgende Metriken zu erfassen:

Codelänge und -struktur	• Lines – Anzahl aller Codezeilen • LOC – Anzahl aller Codezeilen, die Text enthalten • NCLOC – Anzahl aller nichtleeren Codezeilen, die keine Kommentare sind • Prozentualer Anteil an Kommentaren
Codeumfang	• Anzahl der PL/SQL-Dateien in einem Projekt • Anzahl der Pakete in einem Projekt
Codekomplexität	• Anzahl der Funktionen und Prozeduren pro Paket • Absolute und durchschnittliche zyklomatische Komplexität[28] von Funktionen pro Paket

Tabelle 3-5: *Auflistung der im Sonar-Plugin zu erfassenden Metriken*

Die Analysen bezüglich der Codelänge sowie des Codeumfangs sind hierbei obligatorisch, die Untersuchungen zur Codekomplexität sind als optionaler, jedoch wünschenswerter Implementationsschritt zu betrachten.

Neben diesen das Gesamtprojekt überspannenden Untersuchungen sind für das Plugin Konformitätsregeln zu definieren – der Umfang und die Menge dieser Regeln soll sich wiederum entsprechend der zuvor beschriebenen beispielhaften Natur der Implementierung in Grenzen halten.

Zusätzlich zum funktionalen Teil des Plugins sind für alle Teilprojekte und -klassen in angemessenem Maße Tests zu schreiben, die die korrekte Funktion der einzelnen Elemente bestätigen.

[28] auch McCabe-Metrik – Messung der Komplexität anhand von Kontrollstrukturen in Funktionen – http://www.itwissen.info/definition/lexikon/McCabe-McCabe-Metrik.html

4 Implementierung

Im Verlaufe dieses Kapitels werden die schließlich erarbeiteten Lösungen für die gestellten Probleme ausschnitthaft vorgestellt. Hierbei wird auf besondere Teilaspekte sowie sich ergebende Probleme bei der Erarbeitung der Lösungen eingegangen. Zudem wird ein Vergleich zwischen Konzept und letztendlichen Resultaten gezogen und die Funktionalitäten an ausgewählten Beispielen demonstriert.

4.1 PL/SQL-Grammatik

Die Implementierung der DSL-Grammatik, als Anhang zu dieser Ausarbeitung inklusive einer angemessenen Dokumentation auf CD verfügbar, umfaßt etwa 1300 Codezeilen inklusive Kommentaren. Im Folgenden soll nun genauer auf ihre Einzelheiten eingegangen werden.

4.1.1 Struktur

Die Ausarbeitung der Grammatik erfolgte sehr stark konzeptorientiert, wie ihr formeller Aufbau bereits deutlich zeigt. Die Grammatik öffnet neben den Headerinformationen mit einer Spaltung in die Teilsprachen von SQL und PL/SQL sowie der einzelnen, vorher beschriebenen Funktionen, deren Implementation notwendig ist. (Listing 4-1)

```
CompilationUnit:
    statement+=(DDLStatement | DMLStatement | DCLStatement | PLSQLStatement)*;

/* Aufteilung in SQL-Teilsprachen */
DDLStatement:
    (CreatePackage      |
    CreatePackageBody   |
    CreateSequence      |
    CreateSynonym       |
    CreateTable         |
    CreateTrigger       |
    CreateType          |
    CreateView          |
    AlterTable          |
    CommentColumn       |
    CommentTable        |
    GrantStatement      |
    LockTableStatement  )
    lineSeparator?='/'?
;

DMLStatement:
    (InsertStatement    |
    DeleteStatement     |
    MergeStatement      |
    UpdateStatement     |
    SelectStatement     )
    lineSeparator?='/'?
;

DCLStatement:
    (CommitStatement    |
    RollbackStatement   |
    SavepointStatement  )
    lineSeperator?='/'?
;

PLSQLStatement:
    BlockStatement
    lineSeparator?='/'?
;
```

Abbildung 4-1: *Aufteilung der Grammatik in SQL-Teilsprachen*

Es folgen, dem Konzept entsprechend, Datentypdefinitionen sowie nacheinander die Definitionen der einzelnen Funktionen, die Teile ihrer Beschreibungen an sich im nächsten Grammatikabschnitt befindende Hilfsfunktionen und -klauseln delegieren. Diese hierarchische Struktur soll am Beispiel des `GRANT`-Statements verdeutlicht werden (Listing 4-2).

```
210 GrantStatement:
211     'GRANT'
212     (clauses+=GrantPrivilegeClause (','clauses+=GrantPrivilegeClause)*)
213     object=OnObjectClause
214     'TO' grantees+=GranteeClause (',' grantees+=GranteeClause)*
215     ('WITH''HIERACHY''OPTION')?
216     ('WITH''GRANT''OPTION')?
217 ;
...
393 // GRANT
394 GrantPrivilegeClause: {GrantPrivilegeClause}
395     (privileges+=GrantPrivilege | ('ALL''PRIVILEGES'?))
396     ('(' names+=ColumnName (','names+=ColumnName)* ')')?
397 ;
398 GrantPrivilege: {GrantPrivilege}
399     ('ALTER' | 'CREATE''SESSION' | 'DELETE' | 'EXECUTE' | 'INSERT' |
400      'MERGE' | 'REFERENCES' | 'SELECT' | 'UPDATE')
401 ;
402 OnObjectClause:
403     'ON'
404     (   (object=ObjectReference) |
405         (directory=DirectoryName) |
406         (edition=EditionName) |
407         ('JAVA' ('SOURCE'|'RESOURCE') java=ObjectReference))
408 ;
409 GranteeClause: {GranteeClause}
410     (   (user=ID ('IDENTIFIED''BY' password=ID)?) |
411         (role=ID) |
412         ('PUBLIC'))
413 ;
```

Abbildung 4-2: *Hierarchisierung und Delegation im `GRANT`-Statement*

Im `GRANT`-Statement selbst werden lediglich statische Anteile konkret aufgeführt – Dinge, die sich in der Anwendung des Statements stets ändern können, werden unter den Hilfsklauseln, in diesem Falle also der `GrantPrivilegeClause`, der `OnObjectClause` sowie der `GranteeClause` weiter beschrieben. Diese Klauseln teilen sich je nach Komplexität der Funktionen weiter auf, so führt die `GrantPrivilegeClause` aus dem Beispiel schließlich auf eine Auswahl der verschiedenen Privilegien, die durch einen `GRANT` vergeben werden dürfen sowie auf den Datentyp, der Spaltennamen beschreibt (`ColumnName`).

Die weiteren in den SQL-Teilen der Grammatik aufgeführten Funktionen verhalten sich auf ähnliche Weise. Nach diesen Beschreibungen folgt der PL/SQL-Teil der Grammatik – im Wesentlichen werden hier aus Statements Blöcke der prozeduralen Sprache beschrieben. Auch hier findet sich die gleiche hierarchische Struktur wieder – ein allgemeines `Statement` wird in verschiedene speziellere Teilstatements unterschieden, die schließlich einzeln deklariert werden. Listing 4-3 zeigt hinsichtlich dessen auch die Orientierung an der Sprachreferenz und ihren Flußdiagrammen, obgleich es in der Struktur leichte Abweichungen gibt.

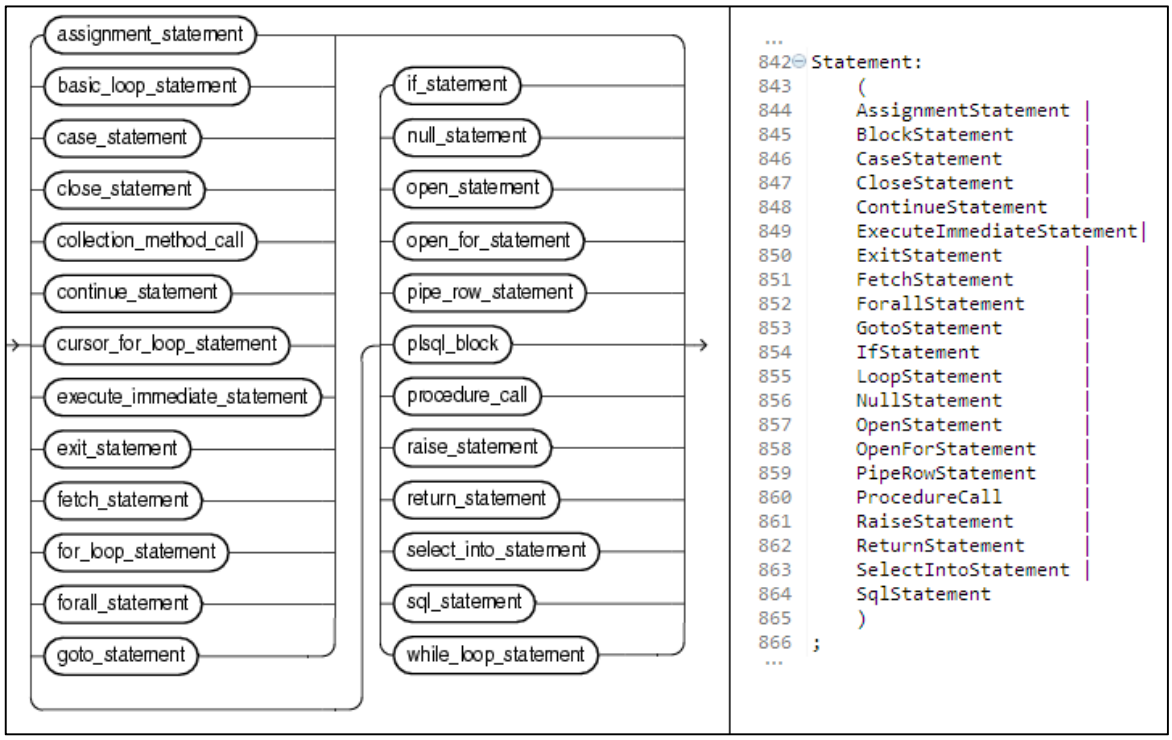

Abbildung 4-3: *Vergleich zwischen Sprachreferenz[29] (angepaßt) und Implementierung*

Schließlich beinhaltet die Grammatik noch den Komplex arithmetischer und datenbanksprachlicher Ausdrücke – auf diese Implementationsdetails wird im nächsten Abschnitt konkret Bezug genommen. Die Grammatik schließt – wie bereits im Konzept beschrieben, mit der Definition der primitiven Datentypen ähnelnden Literale sowie den Terminal-Regeln.

4.1.2 Sprachliche Besonderheiten

Im Rahmen der Grammatik bieten sich einige Besonderheiten bezüglich der Funktionenbeschreibung – schwerwiegendster Punkt ist hier die Arbeit mit **rekursiven Ausdrücken**. Die Schwierigkeit in der Behandlung linksseitiger Rekursionen ergibt sich dadurch, daß Xtext – beziehungsweise der unterliegende ANTLR-Parser – die Grammatik von oben nach unten liest und somit links-rekursive Ausdrücke eine Endlosschleife hervorrufen könnten.[30]

Um diese Probleme zu umgehen, muß durch linksseitige Faktorisierung vermieden werden, daß der Parser dieselbe Regel wieder und wieder aufrufen kann. Solche Situationen finden sich an zwei Stellen in der Grammatik selbst – im `SELECT`-Statement sowie im Komplex allgemeiner Ausdrücke (`Expressions`).

---

[29] Originalgrafik siehe http://docs.oracle.com/cd/E11882_01/appdev.112/e17126/block.htm#CJACJBCH

[30] Expression-Parsing mit Xtext – http://blog.efftinge.de/2010/08/parsing-expressions-with-xtext.html

Die Möglichkeit eines rekursiven Aufrufs im `SELECT`-Statement wird durch mengentheoretische Ausdrücke wie `UNION` oder `INTERSECT` nötig – diese Ausdrücke können zwischen einer Liste beliebig viele `SELECT`-Statements stehen, es kann daher keine simple Beschreibung in der Form `(SELECT […]) UNION (SELECT […])` stattfinden. Zur Lösung des Problems wurden einzelne `SELECT`-Statements als Teile eines gesamten, komplexeren Ausdrucks betrachtet, zwischen denen solche Mengenoperation stehen dürfen (Listing 4-4). Diese Vorgehensweise ist vergleichbar mit einem mathematischen Ausdruck der Form `A+B`, wobei `A` sowie `B` einzelne Zahlliterale sind, die sich mit Operatoren zu einem Gesamtterm zusammensetzen.

```
257 SelectStatement:
258     SelectPartialStatement ({SelectStatement.left=current}
259     (('UNION'|'INTERSECT'|'MINUS') (('DISTINCT'|'UNIQUE')|'ALL')?)
260     right=SelectPartialStatement)*
261
262     orderClause=OrderClause?
263     forUpdateClause=ForUpdateClause?
264 ;
265 SelectPartialStatement returns SelectStatement:
266     'SELECT' (('DISTINCT'|'UNIQUE')|'ALL')?
267     selectList+=SelectList (',' selectList+=SelectList)*
268     into=IntoClause?
269     'FROM' tables+=TableOrJoin (',' tables+=TableOrJoin)*
270     whereClause+=WhereClause?
271     hierQuery+=HierarchicalQueryClause?
272     groupbyClause+=GroupByClause?
273 ;
```

Abbildung 4-4: *Grammatikalische Definition des `SELECT (INTO)`-Statements*

Eine komplexere Implementation solcher Ausdrücke findet sich im Expressions-Komplex der Grammatik. Neben der linksseitigen Faktorierung der Ausdrücke war an dieser Stelle zusätzlich die Reihenfolge der Operationen – das bedeutet also die Stärke der Bindung der Operatoren zu ihren Ausdrücken – zu beachten. Expressions sind so strukturiert, daß Ausdrücke, die am schwächsten gelten, zuerst deklariert und dann in stärker wirkende Operationen eingebettet werden müssen. Um den Begriff der Operationsordnung zu erklären, möge das Statement „`IF 1+2*4=9 AND a.attribute LIKE 'ABC' THEN …`" dienen. Beim Parsen dieses Ausdrucks muß erst aufgelöst werden, welche Elemente in erster Linie zusammengehören – als am stärksten bindende Elemente sind also zuerst die mathematischen Ausdrücke aufzulösen, danach der `LIKE`-Ausdruck und zuletzt die Wahrheitsausdrücke `=` und `AND`. Da diese beiden Ausdrücke auf derselben Ebene stehen, soll der zuerst auftauchende Ausdruck zuerst verarbeitet werden.

Im konkreten Fall bedeutet das also für die Grammatik, daß `BooleanExpressions` zuerst zu definieren sind, da sie schwächer sind als jegliche andere Expression. Dem folgen `InExpressions`, `LikeExpressions`, … bis schließlich die am stärksten verbundenen Ausdrücke wie Literale oder feste Funktionsaufrufe erreicht sind (Listing 4-5).

```
1214 BooleanExpression returns Expression:
1215     LikeExpression (
1216         {BooleanExpression.left=current}
1217         operator=BooleanOperator
1218         right=LikeExpression
1219     )*;
1220 LikeExpression returns Expression:
1221     InExpression (
1222         {LikeExpression.left=current}
1223         ('LIKE'|'NOT''LIKE')
1224         right=InExpression ('ESCAPE' str=StringLiteral)?
1225     )*;
1226 InExpression returns Expression:
1227     Addition (
1228         {InExpression.left=current}
1229         ('IN'|'NOT''IN')
1230         '(' values+=Expression (',' values+=Expression)* ')'
1231     )?;
1232 Addition returns Expression:
1233     StrCombine (
1234         {Addition.left=current}
1235         operator=('+' | '-')
1236         right=StrCombine
1237     )*;
1238 StrCombine returns Expression:
1239     Multiplication (
1240         {StrCombine.left=current}
1241         '||'
1242         right=Multiplication
1243     )*;
1244 Multiplication returns Expression:
1245     PrimaryExpression (
1246         {Multiplication.left=current}
1247         operator=('*' | '/')
1248         right=PrimaryExpression
1249     )*;
1250 PrimaryExpression returns Expression:
1251     Literal                    |
1252     VariableReference          |
1253     FunctionCallWithoutSemicolon|
1254     Negation                   |
1255     Existance                  |
1256     SimpleCaseExpression       |
1257     SelectStatement            |
1258     '(' BooleanExpression ')'
1259 ;
```

Abbildung 4-5: *Implementierung arithmetischer und programmatischer Ausdrücke*

Eine weitere Besonderheit, die die Grammatik beherbergt, ist die „Case Insensitivity“[31] ihrer Schlüsselwörter – das bedeutet, daß beim Parsen der Grammatik keine Rolle spielt, ob ein Befehl in Großbuchstaben (`SELECT`) oder Kleinbuchstaben (`select`) eingegeben wird. Eine solche Freiheit im Schreibstil ist notwendig, da einerseits die Sprachspezifikation keine genauen Regelungen vorschreibt und andererseits in existierendem Programm-

[31] Unabhängigkeit von Groß- und Kleinschreibung

code beide Varianten verwendet werden. Zum Erreichen der entsprechenden Funktionalitäten wurden geringfügig Änderungen am MWE-Workflow der Sprache, im Wesentlichen Ergänzungen um das Attribut `ignoreCase=true` nötig. Das Erlauben einer solchen Funktionalität erschwert jedoch die Grammatikdefinition um ein signifikantes Maß – da der Parser nun jeglich geschriebene Schlüsselwörter als solche liest, müssen Attribute, die sich mit Schlüsselwörtern decken, explizit deklariert werden (Listing 4-6). Zudem wird es unmöglich, Einheitensuffixe wie `T` für `Tera` in der Grammatik zu verankern, da bei der Deklaration eines ‚T' als Schlüsselwort jegliches einzeln stehende ‚t' als solches gelesen werden würde – dieses Verhalten würde es verbieten, Tabellen mit Aliasnamen wie `t` zu bezeichnen oder mit `t.attribut` aufzurufen.

```
1044 // Reservierte Attribute, die explizit erlaubt werden müssen,
1045 // da sie sonst als falsche Schlüsselwörter geparst werden
1046 ReservedAttribute: {ReservedAttribute}
1047     ('TYPE'|'KEY'|'MAX'|'VALUE'|'OPEN'|'CLOSE'|'FLOOR'|'SUM'|'SORT'|'BLOB'|'ENABLE')
1048 ;
```

Abbildung 4-6: *Explizite Deklaration von Schlüsselwörtern als Attribute*

Zuletzt sind als besondere Spezifikation Funktionsaufrufe hervorzuheben – durch die sehr starke Variabilität solcher Aufrufe wurden hinsichtlich dieser Elemente immer wieder Änderungen und Erweiterungen nötig, die alle entsprechenden Aufrufe abdecken. Beispielfälle für korrekte Formen der Anwendung sind `function(attrib).METHODCALL`, `function(attrib1,attrib2)(attrib3)` oder `function(attrib).attribute`. Letztlich wurden in der finalen Grammatik Funktionsaufrufe als Liste einzelner, mit ‘.‘ getrennter Funktionen behandelt, die eine oder mehrere optionale Parameterlisten – mit Parametern, die selbst Funktionen oder Ausdrücke sein können – besitzen können (Listing 4-7).

```
1032 FunctionCallWithoutSemicolon returns Expression:
1033     labels+=Label*
1034     funcs+=SingleFunctionCallWithoutSemicolon
1035     ('.' funcs+=SingleFunctionCallWithoutSemicolon)*
1036     cmc=CollectionMethodCallAttribute?
1037     alias=AliasName?
1038 ;
1039 SingleFunctionCallWithoutSemicolon:
1040     ((func=FunctionCallNative) | (function=FunctionName))
1041     reserved=ReservedAttribute?
1042     (   ('(' ((param+=FunctionParameter (',' param+=FunctionParameter)*))? ')')*
1043     )
1044 ;
```

Abbildung 4-7: *Definition von Funktionsaufrufen*

4.1.3 Umfang und Vollständigkeit

Die Grammatik validiert entsprechend der gestellten Anforderungen mit allen wesentlichen verwendeten Elementen. Ein als Beispiel gewählter PL/SQL-Programmblock wird ohne Probleme validiert - syntaktische Fehler im Programmcode werden jedoch in den meisten Fällen von der Grammatik erkannt und mit entsprechenden Fehlermarkierungen versehen (Listing 4-8). Zusätzlich stellt die Abbildung mit einer Mischung aus klein und

groß geschriebenen Schlüsselwörtern die angesprochene „Case Insensitivity“ der Grammatik dar.

```
declare
  v_summe           number := 0;
  v_summe2          integer := ; -- Leere Zuweisung
  v_auftrag_pos     auftrag_pos%ROWTYPE;
  cursor c_auftrag_pos
    (p_auftrag_nr in auftrag_pos.auftrag_nr%TYPE) is
    select *
      from auftrag_pos
      where auftrag_nr = p_auftrag_nr;

begin
  OPEN c_auftrag_pos (42);
  loop
    FETCH c_auftrag_pos into v_auftrag_pos;
    exit when c_auftrag_pos%NOTFOUND;
    v_summe := v_summe + v_auftrag_pos.anzahl *
                         v_auftrag_pos.preis;
  end loop;
  CLOSE c_auftrag_pos;
end;
```

Abbildung 4-8: *Beispiel eines kleinen PL/SQL-Programms*[32]

Wie bereits in der Anforderungsanalyse angesprochen, umreißt die entwickelte Grammatik die Sprachspezifikation nicht scharf – in Einzelfällen werden Elemente, die der korrekten Syntax nicht entsprechen, nicht als fehlerhaft markiert (Listing 4-9). Im Beispiel dürfte kein String als Grenze der Schleife erlaubt sein[33] – da die Grenzen jedoch an dieser Stelle als Expressions definiert sind und String-Literale als Expressions dieser Definition entsprechen, erscheint an dieser Stelle keine Fehlermeldung.

```
BEGIN
   FOR i IN 5..'Zehn' LOOP
      IF MOD(i,2) = 0 THEN
         INSERT INTO temp VALUES (i, 'i ist gerade');
      ELSE
         INSERT INTO temp VALUES (i, 'i ist ungerade');
      END IF;
   END LOOP;
   COMMIT;
END;
```

Abbildung 4-9: *Darstellung der syntaktischen Unschärfe der Grammatik*

Bis auf wenige Ausnahmen konnten alle gewählten Packages, die als Projektcode im Unternehmen genutzt werden, korrekt validiert werden. In vorherigen Versionen der Grammatik ergaben sich in Validierungstests stets neue Fehler und Ungenauigkeiten, die durch

[32] Beispiel entlehnt aus http://www.datenbank-plsql.de/sql.htm

[33] vgl. http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/loop_statement.htm#i34785 – Abschnitt „lower_bound .. upper_bound“

das im Konzept beschriebene iterative Verbesserungsverfahren schrittweise beseitigt wurden. Eine Beschränkung des Xtext-Tools selbst verbietet es jedoch, Strings ohne Weiteres frei zu definieren; Ausdrücke wie ‚\' oder ‚_' werden in der Standardvariante nicht erlaubt. Das heißt, daß `LIKE`-Ausdrücke nicht in jedem Falle korrekt geparst werden könnten. Zur Lösung dieses Problems muß der `ValueConverter`[34] der Sprache geringfügig angepaßt werden, derartige Ausdrücke zuzulassen – dies geschieht mit einer eigenen Implementation des von Xtext bereitgestellten `STRINGValueConverter` – die Anmeldung des Converters geschieht in der Klasse PlSQLRuntimeModule.java des Hauptprojektes – die Implementationen der entsprechenden Klassen, `PlSQLValueConverter.java` sowie `PlSQLSTRINGValueConverter.java`, befinden sich an der gleichen Stelle. Insbesondere letztgenannte Klasse kommentiert jegliche Änderungen, die im Vergleich zum standardmäßigen ValueConverter nötig waren.

Ein weiteres auftretendes Problem ist die Verwendung von Schlüsselwörtern als Variablenname oder Attribut – ein Beispiel hierfür ist „`key`". Die Grammatik beinhaltet bereits eine Anzahl an Ausnahmen für solche Schlüsselwörter, ist jedoch in diesem Bezug unvollständig. Weiterhin vollführt der Parser in diesem Stadium der Grammatik noch keine Prüfung über die vorherige Definition verwendeter Variablen oder ihre Mehrfachdefinition.

Diese genannten Ansatzpunkte bieten – neben der Implementierung weiterer Funktionen der SQL-Sprache selbst, Ansatzpunkte für eine zukünftige Weiterverbesserung und Erweiterung der PL/SQL-Grammatik.

4.2 Sonar-Plugin

Das implementierte Sonar-PL/SQL-Plugin, ebenfalls auf der der Arbeit anhängenden CD verfügbar, hält sich, ähnlich der zuvor erstellen Grammatik, sehr stark an das ihr unterliegende Konzept. In Bezug auf Struktur und Umfang wurden grundsätzlich alle konzeptuell gestellten Forderungen mindestens erfüllt, teilweise sogar um ein geringes Maß übertroffen. Die folgenden Unterabschnitte führen diesbezüglich konkrete Details auf.

4.2.1 Architektur

Der Aufbau des Plugins erfolgte strikt nach dem vorgelegten Konzept (Listing 4-10). Das heißt, daß eine klare Teilung zwischen seinen verschiedenen Aspekten, wie sie im vorher beschrieben worden ist, stattfand. Zu den drei genannten Teilprojekten kommt in der konkreten Umsetzung das Paket `sslr-plsql-toolkit`, welches im Wesentlichen ein auf

[34] wörtl. Sprachumwandler – zuständig für die Wandlung von Code (insbesondere Escape-Zeichen) für das Parsing und zurück für die Anzeige

SSLR[35] basierendes, simples Programm, das zum Parsing-Test einzelner Dateien – und somit der Prüfung der erstellten Sprachbeschreibung auf Korrektheit – dient, enthält.

Für fast alle der erstellten Klassen existieren zudem entsprechende, mit Hilfe von JUnit[36] erstellte Testklassen, die die Funktionalitäten im Einzelnen prüfen.

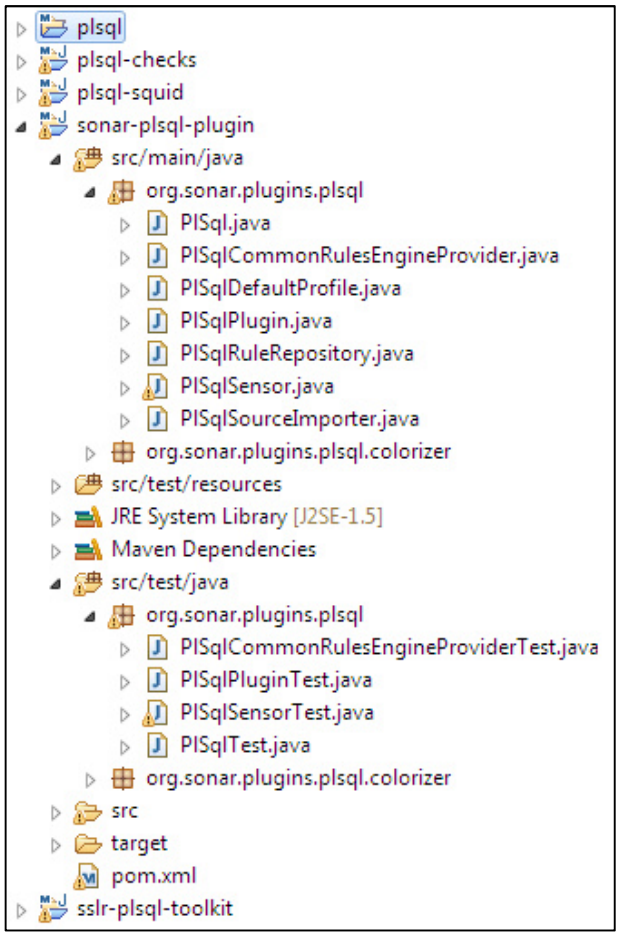

Abbildung 4-10: *Architektur des Sonar-PL/SQL-Plugins*

Auf die Bedeutung und Funktion der einzelnen Pakete wird im folgenden Abschnitt konkret eingegangen.

4.2.2 Implementationsdetails

Um einen tieferen Blick in die Implementierung zu erhalten, werden an dieser Stelle die Klassen der drei für das schließliche Produkt relevanten Projekte mit ihren zusammengefaßten Inhalten aufgeführt (Tabelle 4-11). Die Dokumentation des Programmcodes enthält zu allen Klassen genauere Funktionsablaufbeschreibungen.

---

[35] SonarSource Language Reader – ein von den Sonar-Entwicklern bereitgestelltes Tool zum Parsing-Test

[36] JUnit – ein Framework zum Erstellen von Unit-Tests für Java-Klassen

sonar-plsql-plugin	
`PlSql.java`	Beschreibungen bezüglich der Erweiterungen, für die das Plugin gelten soll, sowie des Kennzeichners für die Sprache selbst
`PlSqlPlugin.java`	Liste der Erweiterungen, die das Plugin enthält
`PlSqlSensor.java`	Auswahl der zu analysierenden Projekte, Erstellen und Abspeichern der Codemetriken und Regelverletzungen
`PlSqlCommonRules EngineProvider.java`	Aktivierung von gemeinsamen Regeln, die die Sonar API für alle Sprachen zur Verfügung stellt
`PlSqlDefaultProfile.java, PlSqlRuleRepository.java`	Festlegungen bezüglich des standardmäßig gewählten Profils für Code-Regeln und Aktivierung der konkreten Code-Regeln
`PlSqlSourceImporter.java`	Delegation auf die Superklasse zum Umgehen eines Bugs, der es andernfalls verhindert, Quelltext nach der Analyse anzuzeigen
`PlSqlColorizer.java, PlSqlDocStringTokenizer.java, PlSqlDocTokenizer.java`	Einfärbung des Quelltexts (Schlüsselwörter, Kommentare) in der Analyseansicht
plsql-squid	
`PlSqlAstScanner.java`	Beschreibung des Scanners, der die Analyse durchführt und die Metriken ermittelt
`PlSqlConfigurator.java`	Konfigurationseinstellungen für den Scanner
`PlSqlMetric.java`	Auflistung aller ermittelten Metriken sowie Konfiguration der Ermittlungsmethoden
`PlSqlCommentAnalyser.java`	Analyse von Kommentaren für entsprechende Metriken (z.B. % der Kommentare)
`PlSqlLinesOfCodeVisitor.java, FileLinesVisitor.java`	Ermittlung der Metriken für Gesamtzeilen und Codezeilen
`PlSqlLexer.java`	Liest Quellcodedateien Token für Token und bereitet sie für den Parser vor
`PlSqlGrammar.java, PlSqlGrammarImpl.java`	Auflistung und Beschreibung von Grammatikregeln und ihren Beziehungen zueinander
`PlSqlKeyword.java, PlSqlPunctuator.java`	Auflistung von Schlüsselwörtern und Begrenzern, die gelext werden sollen
`PlSqlTokenType.java`	Definition spezieller Token der Sprache (z.B. Strings)
`PlSqlParser.java`	Konfiguration des Parsers, Anmelden von Lexer und Grammatikimplementierung
plsql-checks	
`CheckList.java`	Liste aller durchzuführenden Prüfungen bezüglich Codekonformität
`<Name>Check.java`	Testbeschreibungen; Anmeldungen an bestimmte Elemente der Grammatik und Erzeugung der Regelverletzungsmeldungen

`Messages.java`	Externalisierte Sprachstrings der Regelverletzungen zur Gewährleistung der einfachen Umstellung auf Fremdsprachen

Tabelle 4-11: *Übersicht über die Klassen des Sonar-PL/SQL-Plugins*

Die Analyse eines Projektes läuft in der Form ab, daß der PL/SQL-Sensor für das Gesamtprojekt gerufen wird – hier wird ein Scanner erstellt, der die Dateien nacheinander durchgeht und die Metriken erstellt und schließlich aufsummiert. Während der Analyse wird jede einzelne Datei, die eine korrekte Endung und korrekten Quellcode enthält, mit Hilfe des Lexers in Token geteilt, welche vom Parser verarbeitet und in einen AST gewandelt werden. Beim Durchschreiten des Syntaxbaumes werden schließlich an jedem Knoten die entsprechenden Regeln gerufen, die an diesen Stellen angemeldet sind, um die Regelverletzungen zu erzeugen. Der Sensor speichert schließlich die Metriken und Regelverletzungen mit den Zeilen, an denen sie aufgetreten sind, ab, und stellt sie dem Webinterface zur Verfügung.

4.2.2.1 Beispiel: PL/SQL-Lexer

Die Lexer-Klasse **`PlSqlLexer.java`** als Kernstück des Plugins soll an dieser Stelle beispielhaft hervorgehoben und hinsichtlich ihrer Funktionsweise genauer beleuchtet werden (Listing 4-12).

```
public final class PlSqlLexer {

  private PlSqlLexer() {
  }


  public static Lexer create(PlSqlConfiguration conf) {

    return Lexer.builder()

        // Kanäle werden nacheinander gelesen, erster Treffer teilt den gelesenen Token zu
        .withCharset(conf.getCharset())
        .withFailIfNoChannelToConsumeOneCharacter(true)

        .withChannel(new BlackHoleChannel("\\s"))

        .withChannel(commentRegexp("--[^\\n\\r]*+"))
        .withChannel(commentRegexp("/\\*[\\s\\S]*?\\*/"))

        .withChannel(regexp(PlSqlTokenType.STRING,"\'([^\'\\\\]*+(\\\\[\\s\\S])?+)*+\'"))

        .withChannel(new IdentifierAndKeywordChannel(and("[a-zA-Z_]", o2n("\\w")), false, PlSqlKeyword.values()))
        .withChannel(new PunctuatorChannel(PlSqlPunctuator.values()))

        .withChannel(new UnknownCharacterChannel())

        .build();
  }
}
```

Abbildung 4-12: *Lexer-Klasse des Sonar-Plugins*

Der Lexer konsumiert den Code einer Datei Schritt für Schritt, hierbei konsumiert er Token, die den im Programmcode entsprechenden Kanälen entsprechen (`.withChannel`-Beschreibungen). Die Definitionsreihenfolge spielt an dieser Stelle eine große Rolle, der Lexer wird einen konsumierten Token stets dem ersten Kanal zuordnen, für den er zulässig ist.

Bevor dem Lexer Kanäle zugewiesen werden, wird im konkreten Falle zuerst der Zeichensatz der gelesenen Dateien festgelegt, in diesem Falle fast ausschließlich UTF-8, sowie Konfiguration hinsichtlich dessen betrieben, daß der Lexer das Lesen einer Datei abbrechen soll, sobald er ein Zeichen nicht lesen kann.

Die Reihenfolge der Anmeldungen der Kanäle erfolgt schließlich unter dem Gesichtspunkt der vorher erwähnten Arbeitsweise – höherpriore Kanäle sind zuerst anzumelden. Konkret heißt das, daß zuerst alle Whitespaces und Kommentare gelesen werden, danach folgt die Erkennung aller Strings und erst dann die Schlüsselwort- sowie Begrenzererkennung. Letztendlich werden alle noch übrigen Zeichen, als unbekannt konsumiert.

Die Wichtigkeit dieser Anordnung wird beispielsweise ersichtlich, wenn der Quelltext auskommentierten oder in Stringbegrenzern eingefaßten Programmcode enthält – würde der Kanal für Schlüsselwörter und Identifikationsausdrücke in diesem Falle vor den respektiven anderen Kanälen stehen, würde dieser Code als aktiver Programmcode gelesen werden und dem Parser zur Codeprüfung übergeben, obgleich er in der tatsächlichen Anwendung keinen Effekt hat und nicht geparst werden darf.

4.2.2.2 Beispiel: Konformitätsregeln

Als weiterer elementarer Teil der Analyse sind die Beschreibungen der Konformitätsregeln zu sehen – aus diesem Grunde wird eine gewählte, als Modell dienende Regel in ihrer Implementation näher dargestellt. Die Klasse `PackageNamingCheck.java` beschreibt die im vorigen Kapital bereits als Beispiel der Möglichkeiten des Plugins angeführte Regel, daß Pakete in PL/SQL-Dateien stets mit dem Vorsatz `pkg_` beginnen müssen (Listing 4-13).

```
29  @Rule(
30    name = "Package-Name",
31    key = "PackageName",
32    priority = Priority.MAJOR,
33    description = "Verletzung der Namenskonventionen für Pakete")
34
35  @BelongsToProfile(
36          title = CheckList.SONAR_WAY_PROFILE,
37          priority = Priority.MAJOR)
38  public class PackageNamingCheck extends SquidCheck<PlSqlGrammar> {
39
40    @Override
41    public void init() {
42      subscribeTo(getContext().getGrammar().packagename);
43    }
44
45    @Override
46    public void visitNode(AstNode astNode) {
47      if(!astNode.getChild(0).getTokenValue().startsWith("pkg_") &&
48              !astNode.getChild(0).getTokenValue().startsWith("PKG_")) {
49          getContext()
50              .createLineViolation(
51                  this,
52                  Messages.MESSAGE_PACKAGE_NAMING_CHECK, //$NON-NLS-1$
53                  astNode);
54      }
55    }
56  }
```

Abbildung 4-13: *Ausgewähltes Beispiel einer Regelbeschreibung in Sonar*

Regeln in Sonar werden stets zu Beginn der Datei – noch vor der Klassendeklaration – mit Hilfe entsprechender Annotationen als solche deklariert. Die `@Rule`-Annotation mit ihren Attributen beschreibt die einzelne Regel selbst genau und stellt dar, in welcher Form sie schließlich in der Web-Anwendung angezeigt wird, unter der `@BelongsToProfile`-Annotation wird schließlich festgelegt, welchem Profil die Regel zuzuordnen ist.

Die Regelklasse selbst ist üblicherweise in zwei Teile zu unterscheiden – einerseits die Anmeldung der Regel an ein bestimmtes Grammatikelement und andererseits die konkrete Beschreibung der Regel, die zu beachten ist. Die Anmeldung der Regel erfolgt während ihrer Initialisierung, also in der `init()`-Methode und bezieht sich üblicherweise auf ein spezielles Grammatikelement. Wann immer der Parser auf ein solches Element – in diesem Falle also auf einen Paketnamen, stößt, wird die Regel angestoßen und führt ihre Prüfung aus. Hierbei kann die Prüfung zu verschiedenen Zeitpunkten ausgeführt werden:

- beim Eintritt in einen AST-Knoten `visitNode(…)`
- beim Austritt aus einem AST-Knoten `leaveNode(…)`
- beim Eintritt in eine Datei `visitFile(…)`
- beim Austritt aus einer Datei `leaveFile(…)`

Das ermöglicht es, Prüfungen entweder auf die gesamte analysierte Datei oder nur auf einzelne Grammatikelemente zu beziehen.

Die Implementierungen der Prüfungen, die eine Regel durchführt, können in Ausführung und Komplexität sehr stark variieren – die dargestellte Regel vollführt nur eine einfache Prüfung dessen, ob der Wert des ersten Kindes des aktuellen AST-Knotens, also der Name des Paketes, mit `pkg_` oder `PKG_` beginnt. Ist dies nicht der Fall, so wird eine entsprechende Verletzungsnachricht an diesem Knoten kreiert.

Schließlich muß die Regel für den Analysesensor registriert werden – in der Implementation geschieht das in der Anmeldung in der Klasse CheckList.java, welche schließlich alle Regeln gemeinsam übermittelt.

Wie vorher erörtert wurden für jegliche Regelbeschreibungen Unit-Tests geschrieben, die sie auf grundsätzliche Funktionanz und Korrektheit prüfen – anhand einer Datei, die schlechte Praktiken enthält, wird geprüft, ob die Regel für diese Datei mit der korrekten Meldung anschlägt (Listing 4-14).

```
public class PackageNamingCheckTest {

      @Test
      public void test() {
        PackageNamingCheck check = new PackageNamingCheck();

        SourceFile file = PlSqlAstScanner.scanSingleFile(
                new File("src/test/resources/checks/pkg_name_close-if-ineq-linelength.pkg"), check);
        CheckMessagesVerifier.verify(file.getCheckMessages())
            .next().atLine(1).withMessage("Package-Namen sollten mit \'pkg_\' beginnen.")
            .noMore();
      }
}
```

Abbildung 4-14: *Testprozedur für die Package-Name-Regel*

4.2.3 Beispielanalyse

Zur Demonstration der Möglichkeiten und Funktionen des Plugins wurde eine beispielhafte Analyse mit einer Mischung aus unternehmensgenutzten PL/SQL-Paketdateien sowie eigenen Beispieldateien durchgeführt.

Im Vorlauf einer Analyse ist es stets nötig, in der Wurzel des Projektes, das zu analysieren ist, eine Datei mit dem Namen `sonar-project.properties` zu erstellen, welche Analyseparameter wie den Projektnamen, die Orte, an denen die zu analysierenden Dateien zu finden sind oder die Art der Datenbank, in der die Daten zu speichern sind, aufzuführen sind (Listing 4-15).[37]

[37] weitere Informationen zu möglichen Analyseparametern (engl.) – http://docs.codehaus.org/display/SONAR/Analysis+Parameters

```
# Required metadata
sonar.projectKey=plsql.examples.example-project
sonar.projectName=PL/SQL-Beispielprojekt -- Bachelorarbeit
sonar.projectVersion=1.0

# Description of project (optional)
sonar.projectDescription=Ein mit dem Sonar-Runner analysiertes Beispielprojekt

# Comma-separated paths to directories with sources (required)
sonar.sources=src

# Language
sonar.language=plsql

# Encoding of the source files
sonar.sourceEncoding=UTF-8
```

Abbildung 4-15: *Beispielhafte .properties-Datei zur Projektanalyse*

Schließlich ist im Ordner, in dem sich diese Datei befindet, über die Konsole der Befehl `sonar-runner` auszuführen – für eine erfolgreiche Analyse ist natürlich vorausgesetzt, daß der Sonar-Server bereits aktiv ist. Dateien, die nicht korrekt geparst werden können, führen bei der Ausführung des Sonar-Runners zu einer Fehlermeldung und werden bei der Analyse übersprungen.

Nach erfolgter Analyse können die Ergebnisse auf dem Server – der standardmäßige Zugriffspfad, änderbar über die `.properties`-Datei, lautet `http://localhost:9000` – eingesehen werden (Listing 4-16).

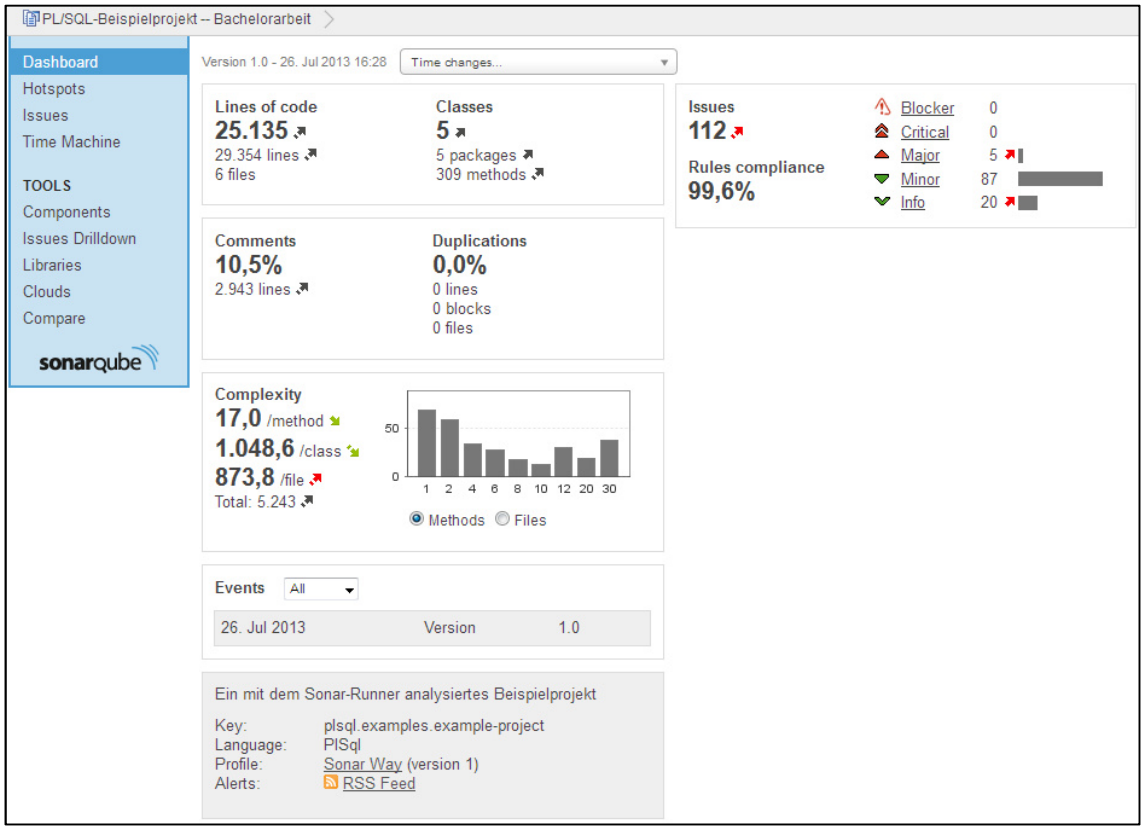

Abbildung 4-16: *Ergebnisse einer Analyse des Sonar-PL/SQL-Plugins*

Die Analyse liefert alle wichtigen Metriken, die in der Anforderungsanalyse und dem Konzept beschrieben worden sind (Codelänge-/struktur, Codeumfang, Codekomplexität), also die quanititativen Analyseaspekte, sowie die Untersuchungsergebnisse eines Grundumfangs eigens definierter Regeln – der qualitativen Codeaspekte.

Da jede Analyse einzeln gespeichert wird, bietet das Plugin neben der reinen Betrachtung auch die Möglichkeit, den zeitlichen Verlauf der relativen und absoluten Konformitätsdaten zu verfolgen – zusätzlich wird im Vergleich zur letzten Analyse eine positive Tendenz mit entsprechenden Pfeilen angedeutet.

Am Interessantesten für die Verbesserung der Codequalität ist jedoch die Möglichkeit des Anzeigens einzelner Verletzungen von Konformitätsregeln im Programmcode selbst (*drill down,* Listing 4-17).

Abbildung 4-17: *Darstellung der Möglichkeit des Issue-Drill Down*

Am Beispiel zeigt sich, wie die zuvor beschriebenen aus teilweise in der Implementation genauer erklärten Regeln ihre Anwendung finden und inklusive der Zeile, an denen sie auftreten, angezeigt werden (Listing 4-18). Zusätzlich wird ein Teil des Quellcode-Kontextes mit dargestellt – dies hat hauptsächlich den Zweck, Hinweise auf Möglichkeit und Notwendigkeit der Beseitigung zu geben. So sind unter anderem Fehler, die in weniger komplexen Kontexten auftreten, leichter zu beseitigen, als solche, die tief im Programmcode verankert sind und so nicht trivial refaktoriert werden können.

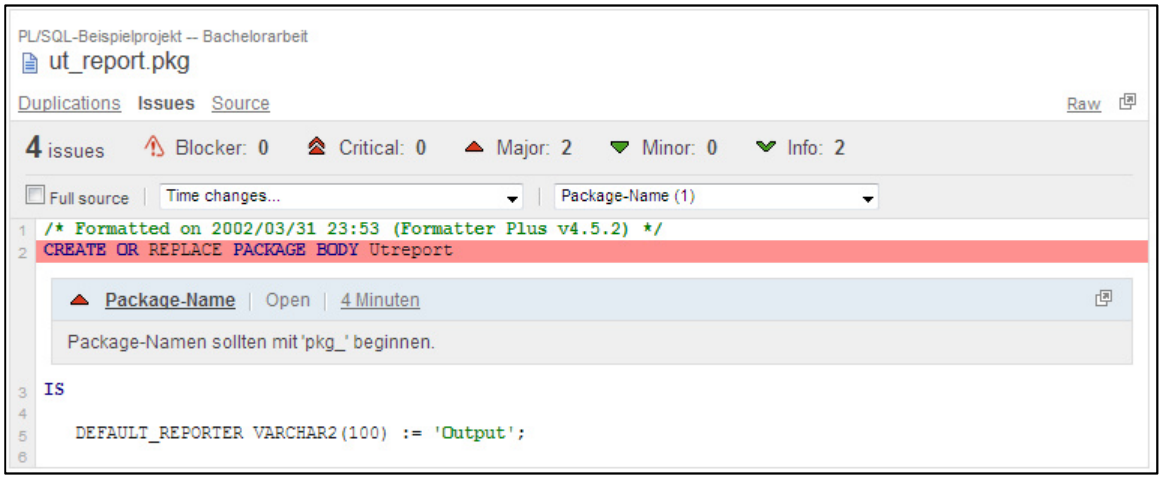

Abbildung 4-18: *Fehlerhervorhebung im Zuge des Issue-Drill Down*

Der Administrator des Plugins hat zu jeder Zeit die Möglichkeit, einzelne Konformitätsregeln zu aktivieren oder deaktivieren, Notizen hinzuzufügen oder ihre Beschreibungen und Parameter zu editieren (Listing 4-19). In nachfolgenden Analysen werden für deaktivierte Regeln keinerlei Prüfungen durchgeführt. Die Anmeldung in den Administrator-Account erfolgt standardmäßig über die Nutzername/Paßwort-Kombination `admin/admin`, der Zugriff auf die Regelprofile und einzelnen Regeln erfolgt über den Link zum Profil auf der Übersichtsseite eines analysierten Projekts.

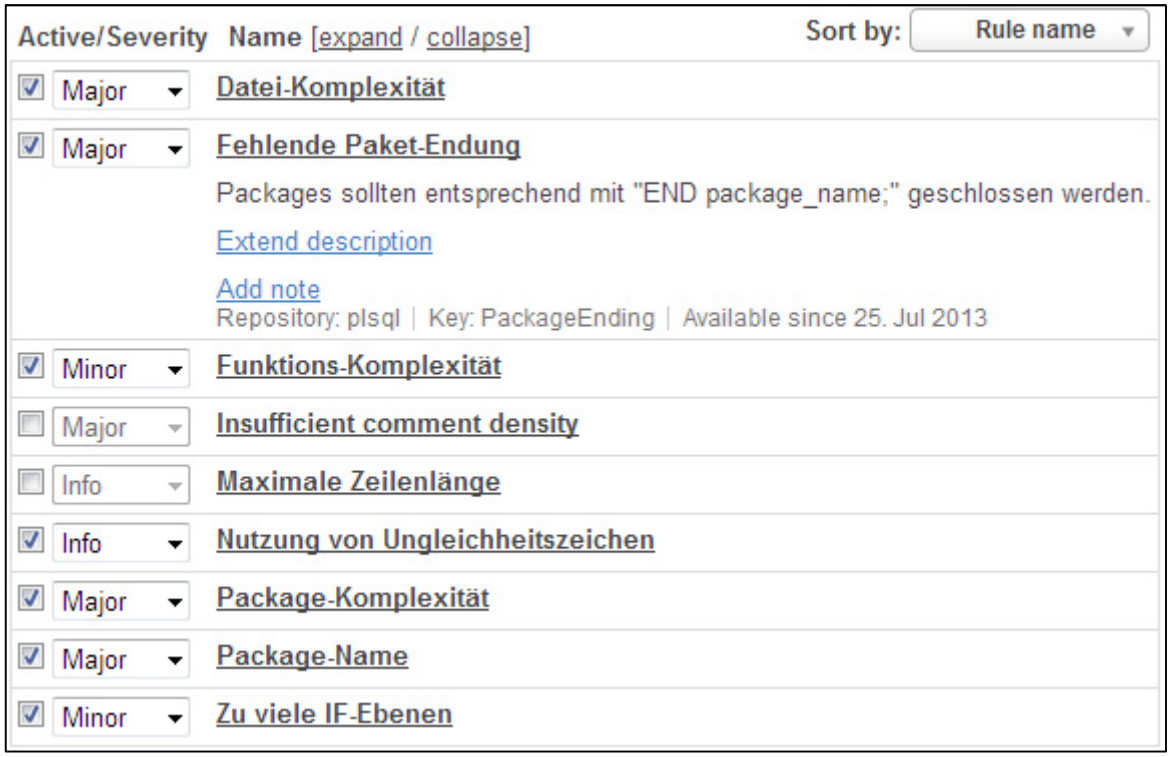

Abbildung 4-19: *Liste der aktiven und inaktiven Regeln des Projektes*

5 Zusammenfassung und Ausblick

In diesem zusammenfassenden Schlußkapitel erfolgt ein Résumé des Erreichten mit einer kritischen Betrachtung der bestehenden Einschränkungen und Verbesserungsmöglichkeiten.

Insbesondere erfolgt ein Ausblick auf mögliche zukünftige Weiterentwicklungsmöglichkeiten der geschaffenen Software.

5.1 Fazit

Zu Beginn dieser Arbeit wurde als grundsätzliches Ziel die Lösung zweier Probleme – konkret der Analyseverbesserung und der Codewartung – vorgestellt. Es ist insgesamt zu sagen, daß mit der Erstellung einer Grammatik für die PL/SQL-Sprache sowie einem entsprechenden Sonar-Plugin diese Probleme zumindest im Grundsatz erfaßt und gelöst worden sind.

Die **PL/SQL-Grammatik** bietet, obgleich in ihrer jetzigen Ausführung noch nicht vollständig, Nutzern bereits eine ausreichend gute Wartungsmöglichkeit für Code. In Ausnahmefällen besteht die Möglichkeit, daß durch eine fehlerhafte Grammatikbeschreibung an einer speziellen Stelle Validierungsfehler auftreten – die Häufigkeit solcher Fehler ist jedoch sehr gering, die Grammatik ist also in ihrer letztlichen Ausführung ein sinnvolles Tool zur prinzipiellen Codewartung. Als problematisch hat sich besonders in späteren Phasen der Entwicklung herausgestellt, daß bei der Generierung der Infrastruktur ein sehr großer Heap-Bereich benötigt wird – in der final enthaltenen Version mußten hierfür 3 Gigabyte bereitgestellt werden; zusätzlich dauerte die Generierung bereits 5 Minuten. Da eine Neuerzeugung bei jeder Änderung der Grammatik nötig ist, ist ein solches Verhalten zunehmend kritisch zu sehen, wenngleich es bei größeren Grammatiken zu erwarten ist und sich bei zunehmendem Wachstum gegebenenfalls weiter verschlimmert.

Das **Sonar-Plugin**, ursprünglich direkt als auf die Grammatik aufbauendes Tool geplant, hat sich während der Entwicklung als mehr oder minder davon isoliertes Projekt herausgebildet – die Xtext-Grammatik fließt in die Sonar-Untersuchung in jetziger Form lediglich marginal als Grundlage der auf Squid basierenden Grammatik des Plugins selbst ein. Nichtsdestotrotz erfüllt die entwickelte Erweiterung ihre Aufgaben weitestgehend – es ermöglicht eine wesentliche qualitative sowie quantitative Analyse des Quellcodes. Ähnlich der unter Xtext erstellten Grammatik kann es beim Parsen des Projektcodes zu einzelnen Fehlern kommen – durch Tests sind jedoch bereits die meisten Fehlerquellen abgedeckt.

Während der Entwicklung der Sonar-Erweiterung hat sich zunehmend herausgestellt, daß die Dokumentation bezüglich der Erstellung weiterer Plugins für das Tool an vielen Stellen unzureichend ist – Schnittstellen zu Sonar-Klassen sowie grundlegende Funktionsweisen sind oftmals schlecht oder gar nicht dokumentiert. Hinzu kommt an dieser Stelle ein mangelndes Supportsystem seitens der Entwickler – es existiert in Form einer Mailing-List[38] die Infrastruktur zur Hilfe, häufig wird jedoch lediglich auf bestehende Lösungen verwiesen, die sich einerseits auf nichtaktuelle Versionen beziehen oder andererseits eine grundlegend andere unterliegende Aufgabenstellung und Architektur besitzen. Diesen Tatsachen geschuldet ist schließlich auch, daß keine Integration der Xtext-Grammatik selbst in das Sonar-Plugin erreicht werden konnte und stattdessen lediglich auf ihrer Basis eine vereinfachte Lösung erarbeitet wurde.

5.2 Ausblick

Wie unter anderem bereits im generellen Fazit angesprochen, existiert für die Lösungen eine Vielzahl an Erweiterungsmöglichkeiten – die entwickelte Software bietet eine erweiterungsfreundliche Infrastruktur für zukünftige Entwickler.

Insbesondere die Grammatik, die unter anderem in der beiliegenden CD zur freien Verfügung steht, kann in ihrer Mächtigkeit stark erweitert werden – so können beispielsweise zusätzliche Funktionen aus dem SQL-Teil der Sprachspezifikation eingeführt werden, Definitionen schärfer gefaßt werden oder existierende Funktionalitäten erweitert werden.

Auch das Sonar-Plugin bietet speziell in zwei Aspekten Erweiterungspotential – einerseits ist das die Beschreibung der Grammatik, die wie angesprochen, stark vereinfacht ist, andererseits ist das die Analysetiefe, also der Umfang an Coderegeln, die analysiert werden. Bei der Untersuchung der erreichten Lösungen haben sich für die Grammatikerweiterung prinzipiell zwei Optionen ergeben, auf die in Zukunft eingegangen werden kann:

- weiterführender Versuch der Integration der Xtext-Grammatik
- Generierung der Squid-Grammatik auf Basis der Xtext-Grammatik

Insbesondere auf die zweite Option ist hier Wert zu legen, da, wie erwähnt, die Schnittstellenbeschreibungen der Sonar-Umgebung an vielen Stellen mangelhaft ausgeführt sind und die Integration so eine sehr große Komplexität annehmen kann. Ein konkreter Ansatzpunkt zur Vereinigung beider Grammatiken ist also in einer Generatorklasse zu sehen, die den Code der Xtext-Grammatik beispielsweise unter Nutzung regulärer Ausdrücke zumindest größtenteils korrekt in den Code der Squid-Grammatik wandelt, so daß eine weitere Entwicklung dort stattfinden kann – eine vollständig manuelle Umsetzung dessen wäre zwar denkbar, würde aber eine sehr große Zeit in Anspruch nehmen.

[38] Weblink zur Sonar-Entwickler-Mailing-List – http://sonar.15.x6.nabble.com/Sonar-dev-f4523654.html

Literatur

[Fega1998] Fegaras, Leonidas – Database Systems II. URL: <http://lambda.uta.edu/cse5331/spring98/plsql.html>, verfügbar am 05.06.2013, 14:26 Uhr

[Fowl2010] Fowler, Martin; Parsons, Rebecca: Domain-Specific Languages. – 1. Aufl. – Amsterdam: Addison-Wesley Longman, 2010

[ITW2013] ITWissen – Online-Lexikon für Informationstechnologie. URL: <http://www.itwissen.info/>, verfügbar am 05.06.2013, 12:37 Uhr

[Oracle2012] Lorentz, Diana; Roeser, Mary Beth: Oracle Database SQL Language Reference, 11g Release 2 (11.2) – 2012. URL: <http://docs.oracle.com/cd/E11882_01/server.112/e26088.pdf>, verfügbar am 13.06.2013, 13:24 Uhr

[Oracle2013] Moore, Sheila: Oracle Database PL/SQL Language Reference, 11g Release 2 (11.2) – 2013. URL: <http://docs.oracle.com/cd/E11882_01/appdev.112/e25519.pdf>, verfügbar am 05.06.2013, 11:37 Uhr

[Sonar2013] Offizielle Dokumentation der Entwickler des Sonar-Tools. URL: <http://docs.codehaus.org/display/SONAR/Documentation>, verfügbar am 19.07.2013, 15:09 Uhr

[TLethen06] Lethen, Tim: Formale Sprachen – Eine Einführung – 2006. URL: <http://www.bezreg-duesseldorf.nrw.de/lerntreffs/informatik/pages/material/sek2/automaten/FormaleSprachen.pdf>, verfügbar am 14.08.2013, 09:45 Uhr

[Xtext2013] Offizielle Dokumentation der Entwickler des Xtext-Frameworks. URL: <http://www.eclipse.org/Xtext/documentation.html>, verfügbar am 11.07.2013, 15:02 Uhr

Anlagen

Anlage – CD-ROM

Auf der der Arbeit beiliegenden CD-ROM befinden sich:

- PDF-Version dieses Dokuments
- Eclipse-IDE (Eclipse Kepler, 4.3) (`/eclipse`) mit entsprechenden Plugins
- Sonar-Server 3.6.2, Sonar-Runner 2.3, Beispieldateien (`/sonar`)
- Quellcodes (`/workspace`)
- Installations- und Nutzungsanleitungen, Weblinks (`README.txt`)

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

Mittweida, den 28. August 2013

_____

Claudio Heeg